Il6r
no.945
cop 2

Math

8

Report No. UIUCDCS-R-78-945 UILU-ENG 78 1741

IMPROVING THE PERFORMANCE OF VIRTUAL MEMORY COMPUTERS

by

Walid Abdul-Karim Abu-Sufah

November 1978 NSF-OCA-MCS77-27910-000036

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-78-945

Improving the Performance of Virtual Memory Computers

by

Walid Abdul-Karim Abu-Sufah

November 1978

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF-MCS77-27910 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, November 1978.

ACKNOWLEDGMENT

Being the student of Professor David Kuck was for me a pleasant encounter of a special kind. I learned a lot from his approach to the solution of problems, from his deep insight into the relevant points of issues, and from his philosophy about things. His advice and guidance were of great value to me while conducting this research. I am grateful to him.

I thank Professor Duncan Lawrie, who was actively involved in different phases of this research. I also thank my friends and colleagues, Utpal Banerjee, Robert Kuhn, Bruce Leasure, David Padua, and Michael Wolfe for the interesting discussions which I had with them. Special thanks go to David Padua; he wrote the trace generator a couple of years ago and helped in getting it running to generate the traces for our experiments. Thanks are also due to Yonsook Kang; she implemented the clustering algorithm. I would also like to thank Mrs. Vivian Alsip for her assistance throughout my stay at the Digital Computer Laboratory and for her help with the typing.

I owe special gratitude to my wife. Going with me through my graduate study was quite an experience for her. I thank her for her patience and love.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

1. INTRODUCTION

1.1 Improving the Locality of Programs - Previous Work

Since the early years of modern computing, people have realized that due to cost-speed tradeoffs, computer memories of very large overall capacity must be organized hierarchically. The introduction of memory hierarchies in computer systems created the problem of storage allocation of programs. At each moment during the execution of a program, the distribution of its information (code and data) among the levels of the memory hierarchy must be determined. The programmer was faced with the additional responsibility of manually solving this memory allocation problem. This was not an easy thing to do, especially with the introduction of high level languages which shielded programmers from the details of machines.

The idea of virtual memory systems was the solution to this problem. It provided an elegant way of achieving automatic storage allocation [KILB62],[SAYR69]. Since the evolution of the virtual memory concept in the early 1960s, a tremendous amount of research effort has gone into investigating the various aspects of virtual memory systems. Different methods of implementation were considered and contrasted: segmentation, paging, or paged segmentation. Moreover different memory management algorithms were investigated. These are concerned with the fetch policy which decides when an item of virtual memory (a page, or a segment) is to be fetched to main memory, the placement policy which decides where to place an item in main memory and the replacement rule which decides which item to replace if there is no space for the new item. Both fixed and variable memory allotment policies were considered [BELA66],[DENN68],[CHU72]. People have used the number of item faults, the efficiency of main memory utilization, and the space-time product cost of a program, to measure the performance of different memory management schemes. Principles of optimality have been defined in [BELA66], [PRIE76], and [BUDZ77]. The performances of different policies were measured by comparisons to the performance of optimal policies. People often use reference string driven simulation techniques for their statistical measurements of the effects of varying memory allotment and page size on the performance of different policies. A survey of the work done in this area and some results can be found in [DENN70], and [KUCK70].

The central reason behind any success which a virtual memory system might achieve is the property of locality of reference which programs exhibit. Denning in [DENN72a] makes the following three statements to describe the locality of reference property of programs:

* During any time interval, a program distributes its references nonuniformly over its address space, some pages being favored over the others.
* The density of reference to a given page changes slowly in time or the set of favored pages changes membership slowly.
* Two disjoint segments of the page reference string tend to be highly correlated when the interval between them

is short, and tend to become uncorrelated as the interval between them increases.

It has been confirmed by early studies that the degree of locality of a program is the most important factor in its cost of execution in a virtual memory computer. Although one may not reduce the number of page faults generated by a program by more than 30 or 40 percent by changing the page replacement algorithm [BELA66], an improvement of a factor of 5 was achieved by improving the locality of programs [COME67]. Thus it was recognized that efforts should be directed to develop techniques to improve the locality of programs before executing them in virtual memory systems. This was an absolute necessity for certain kinds of programs, namely those processing large multi-page arrays.

There are two approaches to the problem of improving the locality of reference strings generated by programs. In the first approach the programmer was expected to follow certain rules and guidelines when coding the solution to different problems. In the second approach people tried to devise automatic or semi-automatic locality improvement techniques. In the following two sections we will discuss briefly the previous work done in these two areas. We will give illustrative examples and sample results. In Section 1.2 we will point out the deficiencies and problems in the previous work, present our philosophy and approach to the problem, and finally sketch the outline of this thesis.

1.1.1 Programmer Implemented Locality Improvement Techniques

It did not take too much time for people to realize that virtual memory computers did not relieve the programmer completely from worrying about the memory needs of a program. When programmers worked under the assumption that in a virtual memory computer they could get all the memory space they needed, the costs of running some programs were high [FINE66], [BRAW68],[GLAS65].

Several papers have been published to give programmers rules and guidelines when writing code to solve large problems in a virtual memory computer. Some of these papers were oriented towards specific applications and problems, others were of more general nature. Examples of the problem oriented work can be found in [BRAW70],[BOBR67],[DUBR72], and [ROGE73], which treat sorting, list processing, solution of eigenvalue problems, and the solution of linear equations respectively. [McKE69],[MOLE72], and [ELSH74] are examples of papers which address the general problem of algorithms for large matrix programs in a paging environment. Moreover, manufacturers of virtual memory computer systems started to devote sections of manuals to help programmers develop a programming style for virtual storage systems [IBM73].

A good representative of this approach to improve program locality is the work of Elshoff in [ELSH74]. He was concerned with the processing of multi-dimensional arrays in a paging environment. In particular he considered two dimensional arrays which were assumed to be stored row-wise. An NXN matrix satisfied the relation $N \leq Z \leq N^2$, where Z is the page size. Elshoff presented some rules to be used by programmers when writing code to solve matrix problems. He applied his individual rules and their combinations to two example programs, namely matrix transpose and matrix multiplication. He also derived analytical expressions for the number of generated page faults when executing under an LRU page replacement algorithm. Moreover, he executed the original programs and the improved programs on a dedicated machine. The matrices were square matrices of size 101x101, each spanning 20 pages of virtual space with a page size of 512 words. Figure 1-a and Table 1 show the results for the matrix transpose program. Figure 1-b and Table 2 show the results for the multiplication program.

There are two very important conclusions which one can make by examining these figures and tables. The first is that programs which process large arrays of data can have very serious problems if executed in virtual memory computers. The second is that the amount of improvement which was attained by the suggested techniques is very significant.

1.1.2 Automatic or Semi-Automatic Locality Improvement Techniques

The main attractive feature of virtual memory systems is the automatic management of memory allocation. Hence the approach presented in the previous section seems to be a step backward, since the programmer is required to follow certain rules while programming for a virtual memory computer. Many of the programming guidelines are either problem oriented or cannot be applied in simple, direct ways to complex and large programs. Hence it seems that if anything is to be done to programs to improve their locality properties, it should be taken care of by the computer software system and not by the programmer.

Several people took this approach [COME67],[HATF71],[MASU74], and [FERR74]. All these researchers worked on what is called the 'pagination problem'. A program has a number of modules: main procedure, subroutines, and data blocks. Assuming that a page can hold more than one module, the pagination problem can be simply stated as trying to group these modules or blocks in pages such that the program generates a more local reference string when executed in a virtual memory computer. Thus the aim is to

Table 1.

Results for the Matrix Transpose Program [ELSH74].
(Memory Allotment 15K)

Algorithm Used	Problem CPU	System CPU	Total CPU	Elapsed Time	I/O Time
Standard	.819	9.900	10.719	77.5	66.8
Combination of All Improvement Rules	1.110	1.408	2.518	11.0	8.5

Table 2.

Results for the Matrix Multiply Program [ELSH74].
(Memory Allotment 15K)

Algorithm Used	Problem CPU	System CPU	Total CPU	Elapsed Time	I/O Time
Standard	197.3	4493.9	4691.2	19460.	14768.4
Combination of All Improvement Rules	222.7	6.9	229.6	252	22.7

Units are seconds

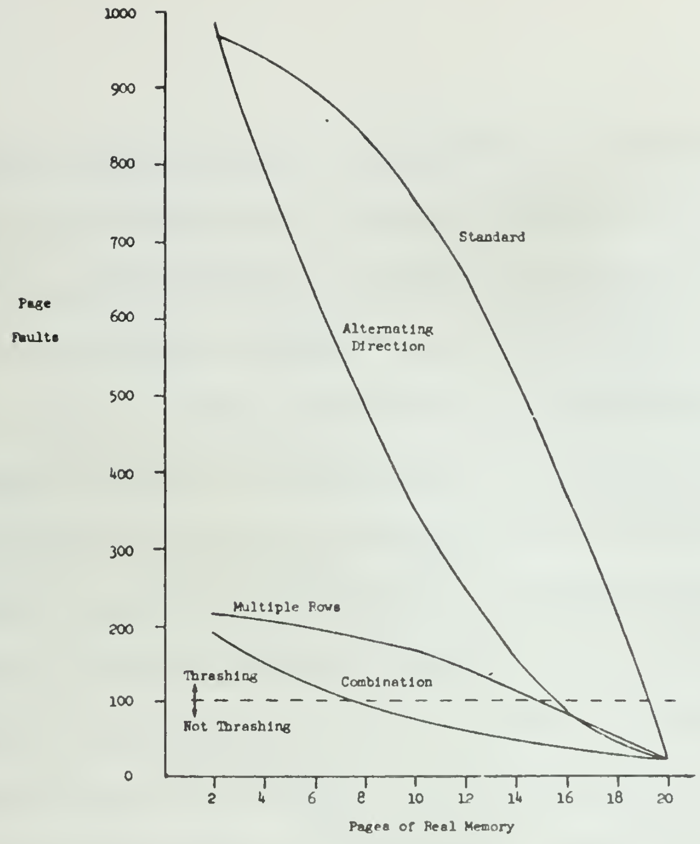


Figure 1-a. Comparison of Matrix Transpose Algorithms [ELSH74]

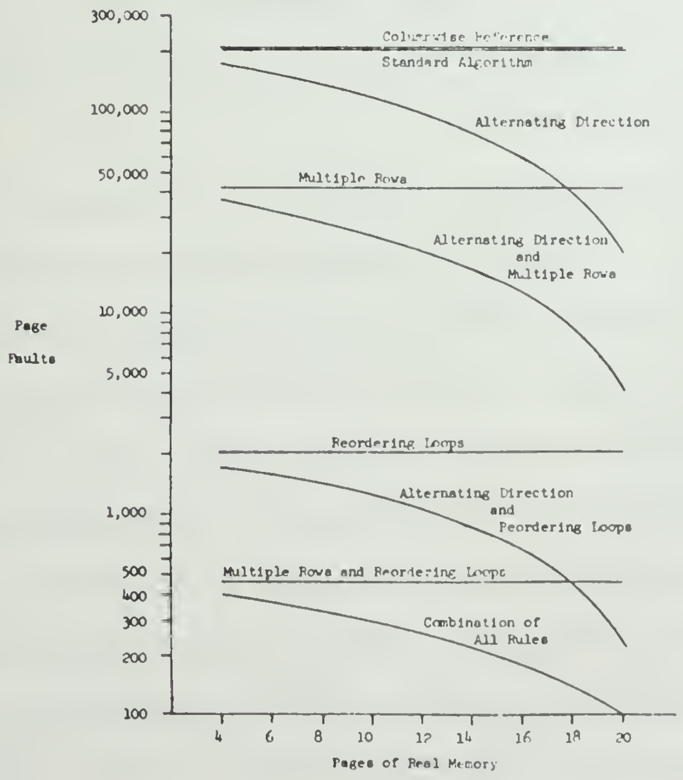


Figure 1-b. Comparison of Matrix Multiplication Algorithms [ELSH74]

modify a program's layout in virtual space. This is called "program restructuring." If the program's modules are relocatable with respect to each other, this can be done by relinking the modules after changing the order in which they are presented to the linker, otherwise changes in the source code and recompilation of some modules might be needed. Information about the dynamic behavior of the program is gathered during an information gathering run. This information is used to construct a restructuring non-directed graph for the program according to a particular restructuring algorithm. The nodes of the graph represent the modules of the program. The numerical labels of the edges represent the desirability that the nodes they connect be laid out together within the same page. After the restructuring graph is constructed a clustering algorithm is used to obtain the new layout for the program from the graph. The clustering algorithm aims at "determining a linear arrangement of nodes (of the restructuring graph) in pages which maximize the vicinity of those pairs having the highest labels" [FERR76b].

The main difference between researchers in this area is the restructuring algorithm they used. Hatfield and Gerald introduced the nearness method for a restructuring algorithm [HATF71]. They argued that performance can be improved if consecutive blocks or modules in the block reference string generated by a program were grouped in the same page. Hence the label E_{ij} of the edge connecting nodes i and j in the restructuring graph, is incremented by one every time block i is referenced directly after j or block j is referenced directly after i. In their extension to the nearness method, Masuda, Shiota, Noguchi, and Ohki [MASU74] incremented E_{ij} if references to i,j are separated by some small distance in time. Ferrari in

[FERR74],[FERR75],[FERR76a], and [FERR76b] takes into explicit account the memory management policy of the system when designing the restructuring algorithm. He argues that each page replacement policy assumes a certain model of the ideal program behavior, which is the behavior of a program for which all the predictions made by the policy are correct. Hence a program is restructured such that its behavior is as predictable by a certain policy as possible. Thus he introduced different program tailoring algorithms for different memory management policies: the critical LRU restructuring algorithm for the LRU replacement policy, the critical working set restructuring algorithm for the working set policy and so on. In the working set policy, for example, the block reference string and the knowledge of the window size T of the working set, $W_b(t,T)$, allow us to identify the blocks which will be in memory at each reference of the string. The critical working set tailoring (restructuring) algorithm increments by 1 all the labels of the edges (in the restructuring graph) which connect a critically referenced block (a block which is not in $W_b(t,T)$) to all the nodes of the members of W_b at the time the critical reference is issued. Ferrari experimented by applying his algorithms to a collection of programs. Some of his experimental results are shown in Table 3. The cost of the restructuring algorithms in terms of computer time varies roughly linearly with the number of references in the string to be examined. The cost of the clustering algorithm (i.e., determining which nodes of the restructuring graph should be grouped in one page) increases less than quadratically with the number of nodes in the restructuring graph. One notices that the data collection which is needed for restructuring is expensive and difficult in today's systems. Restructuring was recently implemented on the SIRIS 8

Table 3.

Results of Program Restructuring Experiments [FERR76b].

Restructured Program	Restructuring Algorithm	Memory Policy	Page Fault Rate Reduction Factor	Mean Working Set Size Reduction Factor
AED compiler	Nearness	LRU, FIFO, Random	2-4	1.1-1.25
Fortran compiler	(MWS)	-	-	1.5-1.8
Interactive editor	Nearness	Pure working set	1.56	-
	CWS	Pure working set	1.86	-
File system	Nearness	Pure working set	2.32	-
	CWS	Pure working set	3.60	-
Fortran compiler	CWS	Sampled working set	2.1-5.5	1.15-1.42
	CSWS	Sampled working set	3.2-12	1.08-1.34
Application program	CWS	Sampled working set	1.4-16.2	0.79-0.99
	CSWS	Sampled working set	3.8-20.8	0.85-1.21
Pascal compiler	CSWS	Sampled working set	1.2-2.4	1.18-1.22
Fortran compiler	CLRU	LRU	1.1-1.9	-
	CFIFO	FIFO	1.1-2.0	-
Simulator	CWS	Pure working set	1.7-3.2	1.06-1.42
	MWS	Pure working set	1.1-1.75	1.22-1.54
Fortran compiler	MWS	Pure working set	1.16-1.32	1.3 -1.65
Application program	MWS	Pure working set	1-3	1.13-1.80

operating system in France. A reduction of 40% to 70% in the page fault rate was reported [BABO77].

1.2 Problems with Previous Work and Our Approach

It is clear from the discussion presented in the previous section that the locality of almost any program can be improved in one way or another. This leads to the conclusion that most programs are not naturally and by their intrinsic properties well suited to run in a virtual memory system. In fact the very early experimental evaluation studies of virtual memory systems did point out that if these systems are going to achieve an excellent level of performance then it must be assumed that the system software of the machine or the programmer will do the work necessary to adapt programs to virtual memory systems [FINE66]. These studies have shown that programs which are written without paying any attention to the paging problem tend to need a large fraction of their virtual space in main memory in order to execute efficiently. This reduces the effectiveness and advantages of virtual memory systems.

We adopt the point of view that the locality improvement work should be done automatically by special software facilities whether these are separate from the rest of the system software of the machine or integrated into some parts of it. Thus the central drawback of all the work presented in Section 1.1.1 is that it puts the burden of locality improvement on the programmer. The program restructuring approach of automatically improving the locality of programs suffers from its limited scope of applicability. The main assumption which is made in the restructuring approach is that the individual modules of a program are smaller than the page size. This is true for code modules and data modules of programs handling small aggregates of data like scalars or small arrays. It is not true however of many practical programs. The size of a data block can easily exceed the size of a page. For example the size of a 32x32 double precision matrix is 8 kilobytes which makes 2 pages of the IBM 370/158 virtual memory space, and arrays are often much larger than 32x32. There are numerous scientific application programs in which tens of large arrays are used. Elshoff's measurements give a hint of the very poor performance which will result if these programs are run on virtual memory computers. The problem will be much worse in the future, because as the CPU speed grows from one computer model to its successor, people will improve the models they are using in their programs because of the better computational power they have available. This will definitely blow up the array sizes used in programs. One can argue that main memory is getting cheaper every day, machines will have more memory attached to them, and hence the problem will not be so bad. The counter argument is that however cheap memory and I/O devices are going to become, they will remain the most expensive parts of a computer system. So the question we face is one of cost-effectiveness.

We summarize the previous discussion as follows. Examining the amount of improvement which Elshoff was able to get, one concludes that virtual memory computers cannot survive without doing something to the kind of programs which Elshoff worked with. Moreover, it is clear that the restructuring approach will not help these programs much. The data blocks in such programs are simply much larger than a page size and thus the problem of which blocks should be grouped in the same page is not the core problem here.

The purpose of this thesis is to provide algorithms for automatic localitity improvement techniques which can be used in an optimizing compiler when compiling programs with large (multi-page) arrays. In Chapter Two we will explore some of the theoretical fundamentals behind this subject. We will discuss concepts like performance measurement criteria and modeling of program behavior. In Chapter Three we will present our transformation algorithms. In Chapter Four we discuss some experiments which we performed on a collection of Fortran programs. In these experiments we evaluated the amount of improvement which was achieved by applying our transformations to these programs using the LRU and the working set memory management policies. We show that the amount of improvement achieved is comparable to the improvement achieved by Elshoff by his programmer implemented techniques [ELSH74]. In several of our programs we encountered the working set anomalies as described in [FRAN78]. We have done some experiments to investigate this anomalous behavior of the working set policy. We also did some experiments which are related to the problem of modeling of program behavior. We conclude this thesis in Chapter Five by pointing out some interesting problems for future research.

2. FUNDAMENTAL CONCEPTS

In this thesis we are concerned with the paging problem of scientific programs which handle large aggregates of data in the form of vectors or multi-dimensional arrays. Due to the nature of such programs, their paging activities will be mainly dominated by data paging of arrays. Hence, in our study we will ignore memory references to scalar variables. Moreover, we will ignore memory references to instructions. Although we believe that our locality improvement techniques will also improve the locality of references to code pages, we will simplify our discussion by separating code and data pages and concentrate on analyzing data paging. Since data paging dominates the I/O activity of the type of programs we are interested in, ignoring code paging does not affect the accuracy of our results in any significant way.

We start this chapter in Section 2.1 by a brief discussion of performance measurement criteria of paged virtual memory systems. In Section 2.2, we will address the modeling problem of program behavior. A survey of the previous work is presented in Sections 2.2.1 and 2.2.2. Traditionally, people were concerned with modeling the locality property of reference strings. In Section 2.2.3, we will present our own different point of view. We will be concerned with identifying localities at the source program level. All the properties of reference strings can be attributed to source program structures. In Section 2.2.3.1, we develop the elementary loop model (ELM) of program localities. We will present examples of loops which follow this model. In Section 2.2.3.2, however, we show examples of loops which cannot be modeled by this model. Nevertheless, such loops can be transformed such that they will follow the ELM. The required transformations are part of those discussed in Chapter 3.

2.1 Criteria for Performance Evaluation

Since the purpose of our work is to improve the locality of programs, we need to define some measurement tools to be used for the evaluation of the degree of locality of programs. Several of these tools can be defined. There are two main categories of these measurement criteria.

In the first category, one measures the intrinsic characteristics of a program, irrespective of the type of machine environment where this program is to run. In other words, the characteristics of program locality intervals are the criteria to be used. Although there has been no general agreement on the definition and the method of isolation of localities of a program, the important characteristics of these localities which determine the cost of running the program in different environments can easily be recognized. The first characteristic is the amount of memory required by each locality. The second is the length of time the program will stay in this locality. These characteristics are called the size of the locality set of pages and the duration of the locality interval. Thus, to compare two programs, one says that the program with the smaller and longer locality intervals is a better program. Moreover, the manner in which the program moves from one locality to another is important. More I/O activity will be generated when adjacent localities have very few common pages.

Another way of measuring the locality of a program is by measuring the cost of its execution in a virtual memory computer. Here one needs to differentiate between monoprogrammed-dedicated systems and multiprogrammed-general purpose systems.

In the monoprogrammed case, the program is allocated all the users' primary memory space of the machine. Thus, the cost of running the program is proportional to the time it spends in the system, or its turnaround time. This in turn is dependent on the amount of time the CPU is used and the amount of time the I/O channels are utilized. In simple words, the turnaround time is given by the equation (assuming paging is done on demand only)

$$\text{Turnaround time} = \text{CPU time} + \text{I/O time}.$$

The I/O time is totally dependent on the degree of locality of the program. The CPU time is mainly dependent on the amount of calculation performed by the program. The efficiency of any technique which is to improve the locality of a given program is measured by the ratio of turnaround time of the original program to that of the transformed program. If the transformation technique does not change the CPU time in any significant way, then the ratio of the I/O time of the original program to that of the transformed program is the measure. This ratio also reflects the improvement in the throughput of a monoprogrammed system. Hence, the better the locality of programs, the higher the throughput of a monoprogrammed system.

The analysis of multiprogrammed systems is more complex.

We must make some assumptions in order to use the cost of execution of programs in a multiprogramming environment as a measure of their degree of locality. On an abstract level, one can say that in a multiprogrammed system there are three resources: CPU bandwidth, main memory bandwidth, and I/O bandwidth. The CPU bandwidth reflects the computational capability of the system, the main memory bandwidth reflects the size and speed of the main memory, and the I/O bandwidth reflects similarly the size and speed of I/O devices and peripherals. In order to use the cost of execution as a degree of locality measure, we must assume that the system is totally saturated. In other words, the CPU, main memory, and I/O channels are 100% utilized. If this is the case, then a program will be using the CPU with a portion of its virtual space present in main memory. The rest of the main memory will be occupied by other programs that are doing I/O or waiting for I/O or CPU service. When the running program references a page which is not in main memory, it loses the CPU to another program and T time units will pass before it gets hold of the CPU once more. T, the reactivation time, is the sum of the I/O time and the system overhead time necessary to service a page fault. A program will be charged by the system as long as it is occupying some part of the main memory, whether it is using the CPU or not. Hence, the cost of executing a program under such conditions is proportional to the time integral of the main memory space it is using at any instant of time over its total life time in main memory. This integral is called the space-time cost.

A multiprogrammed system can use one of several memory management policies. If the local LRU replacement algorithm is used, the program will be assigned a fixed number of page frames all through its lifetime in main memory. When a page fault occurs, only one of the program's own pages will be replaced. Thus, the space-time product cost for a fixed memory allotment is given by:

space-time cost = m * (T*PF + tp) page frames-seconds.

m = # of page frames allocated.

T = average reactivation time in seconds.

PF = # of page faults during the program's lifetime.

tp = the time period in which the CPU was used by the program.

If a variable memory allocation policy is used like the page fault frequency replacement algorithm [CHU72] or the working set policy [DENN68], then the program will go through a sequence of states S_1, S_2, ..., S_i, ..., S_k. The program will stay for t_i seconds in state S_i and will have m_i main memory page frames assigned to it during S_i. Hence, for variable memory allotment we have:

space-time cost = $\Sigma m_i * t_i$, for all i.

One can measure the degree of locality of a program by the inverse of its space-time cost. Hence, another way of measuring the effectiveness of a transformation technique in improving the locality of a program is by measuring the ratio of the space-time cost of the original program to that of the transformed program.

This will also be proportional to the improvement in the throughput of the system. If the reduction of the space-time cost is accompanied by a reduction in the average number of page frames assigned to the program during its lifetime, then an improvement of the degree of multiprogramming will also be achieved.

We will use all these different criteria to measure the improvement of the behavior of programs in different environments. For monoprogrammed systems we will use the number of page faults generated as a function of memory allotment. For multiprogrammed systems, we will use the space-time cost as a function of average memory allotment. We will also consider the intrinsic characteristics of program localities; namely, the size of the locality set, its lifetime, and the transition behavior from one locality set to another. In the next section we will clarify the concept of locality and define the characteristics of a locality from the source code structure of the program.

2.2 Modeling Program Behavior

Although the term "program behavior" has broad implications, it is usually used to mean the behavior of page references of programs. Here we will also restrict ourselves to this specific aspect of program behavior. The page referencing behavior is very important in all computer systems analysis and simulation studies. There have been two methods of analysis of computer systems; namely, mathematical queuing models and simulation models. In the mathematical models, people have been using simplified and inaccurate models of program behavior. In simulation studies, people use traces of actual programs to drive their models. There are several drawbacks to the use of reference strings in simulation studies. Because it is very expensive to generate reference strings, people experiment with a very small number of programs--in most cases, about five programs or so. Their programs may not be truly representative of typical programs. In many cases, it is difficult to extrapolate the behavior of the experimental programs to similar programs. Another drawback of reference strings is that they may contain more detail than is necessary for accurate system modeling.

Thus, there is an obvious need for accurate models of program behavior. These models will replace real program traces in simulation studies. They will be used to generate reference strings for these studies. The length of the reference string generated by these models can be of any desired length. Another advantage of a model over an actual reference string is that it can be used in analytic studies while a reference string cannot. Moreover, the model does not need any large storage space like a reference string.

There have been several efforts to develop such models. All people who worked in this area looked upon the reference strings generated by programs as the observed phenomenon to be modeled. The property of concern of these strings is the locality property. Two types of models have been suggested: stochastic models and deterministic models. We discuss the previous work in these two areas in the following two sections.

2.2.1 Previous Work - Stochastic Models

Different stochastic models have been proposed. A detailed discussion of these models can be found in Spirn's book [SPIR77]. The most important model is the LRU stack model and its extensions [DENN72b], [ARVI73], and [SHED72]. For a reference string $r_1, r_2, \ldots, r_t, \ldots$ at any time t, the LRU stack is an ordered vector $P(t) = (P_1(t), P_2(t), \ldots, P_i(t), \ldots, P_n(t))$ where n is the number of pages in the program and $P_i(t)$ is the identifier of the ith most recently referenced page at time t. For the reference string $r_1, r_2, \ldots, r_t$ there is a corresponding distance string $d_1, d_2, \ldots, d_t$. If $P(t-1) = (P_1(t-1), P_2(t-1), \ldots, P_i(t-1), \ldots, P_n(t-1))$ and $r_t = P_i(t-1)$ then $d_t = i$. In other words r_t is at distance d_t in $P(t-1)$. In the simple LRU model each distance is assigned a probability:

$$P_r[d_t = i] = a_i \quad , \quad 1 \leq i \leq n.$$

In order for the LRU model to exhibit the locality property, it should satisfy the condition:

$$a_1 \geq a_2 \geq \ldots \geq a_n.$$

This locality condition has been shown to be approximately true for real programs [DENN72b]. The distance probabilities can be determined from measurements on real programs. In the distance string $d_1, d_2, \ldots, d_k$ corresponding to a reference string of a program one can count the number of occurrences of a certain distance i, then

$$\hat{a}_i = \text{maximum likelihood estimate of } a_i = (\text{number of occurrences of distance } i)/k.$$

The problem with this method is its expense and that there is no obvious way of "perturbing" these measurements to model other strings. Empirically it was found that approximations to the a_i's can be derived from Belady's lifetime function [BELA69]:

$$A_i = a_1 + a_2 + \ldots + a_i \approx 1 - c\, i^{-k}, \quad 1 \leq i \leq n, \quad 1 < k < 3.$$

Although the simple LRU model did produce good predictions of the average working set size and page fault rate of some real programs in [DENN72b], it fails to predict all aspects of realistic program behavior. For example, in real programs page faults tend to occur in clusters. This happens when a program enters a new phase of execution. The LRU model does not predict this clustering effect [DENN75]. In a memory of m page frames, the probability of a page fault under LRU replacement algorithm is given by

$$\Pr[r_t \notin P(t-1)] = a_{m+1} + a_{m+2} + \ldots + a_n = 1-A_m.$$

This is a constant probability all through the execution of the program. The time until the next page fault is not affected by the number of faults that occurred recently.

The simple LRU model suffers from another problem. It can be shown that for LRU stack model programs, the page fault rate under static LRU is better than that for a dynamic algorithm with the same average size [SPIR77]. This is in contradiction with the experimental evidence available in literature that, for example, the working set algorithm performs better than LRU. The LRU model assumes that the size of the locality set is fixed while in real programs it is varying.

There have been several attempts to improve the simple LRU model. To account for clustering of page faults during phase transitions, the distance probabilities must be allowed to vary in time. Thus we will have

$$a_{1,t} \geq a_{2,t} \geq \cdots \geq a_{n,t} ,$$

for all t, but in general $a_{i,t} \neq a_{i,t+1}$. In a simplified analysis one would assume that there are two distributions of the distance probabilities. One represents the intraphase behavior and is biased toward the top of the stacks. The second corresponds to phase transition behavior and is biased toward larger stack distances. A two-state Markov chain can be used to choose between the distance distributions. This is shown in Figure 2. In state 1 the intraphase distribution is used. In state 2 the phase transition distribution is used. 1-p is the probability of making a phase change and p is the probability of staying in the same phase. $p >> q$ because programs do not spend much virtual time in phase transitions. Although this two-distribution model exhibits the clustering of page faults and phase transition phenomenon, it does not allow for changes in a program's locality set size. This requires a distribution for each locality set size and possibly more than one distribution to model phase transitions. The multiple-distribution model is complicated, impractical, and attempts at validating it have been unsuccessful.

Other Markovian models have been discussed in the literature [SPIR77] and [SHED72]. There are several problems with many of

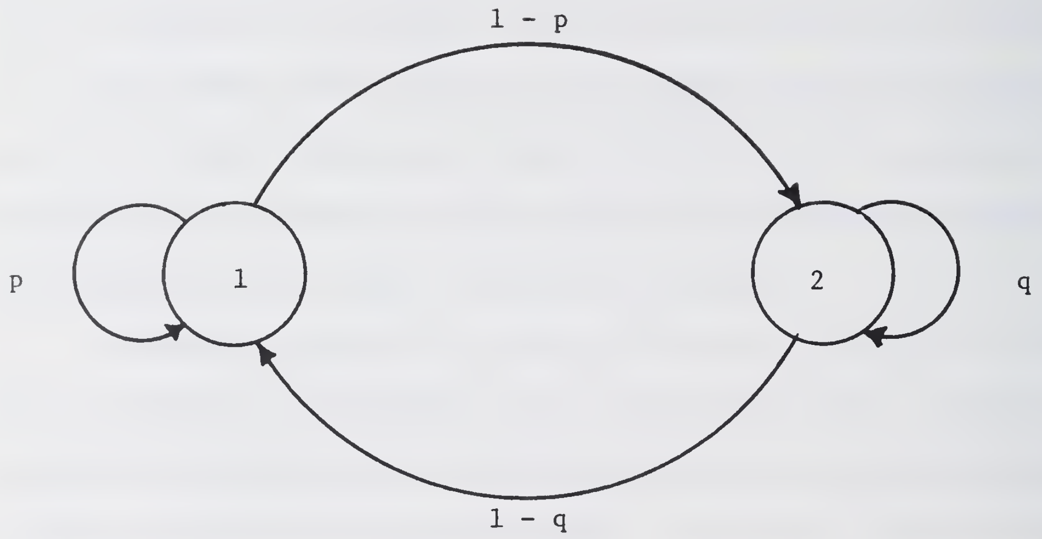


Figure 2. Two-State Markov Chain

them. Mainly these problems are validation, complexity, and practicality problems. The stochastic approach to modeling program behavior seems to go in a vicious circle. If the proposed model is simple and practical, it is not accurate. On the other hand, if more accuracy is incorporated in a model, it becomes complex, impractical, and difficult to validate. We choose to end the discussion of stochastic models at this point and refer the reader interested in more details to [SPIR77].

2.2.2 Previous Work - Deterministic Models

As was mentioned previously, the locality property is the central property of reference strings which everybody is trying to model. In all the literature dealing with stochastic models of program behavior, people talk about the locality property in a vague manner. People argue that at any moment of time t, there exists a set of favored pages which the program tends to reference for a long period of time. This set is called the locality set and the time which the program spends referencing its member pages is called the residence time in the particular locality set [DENN72a]. Thus the program will go through a sequence of states $S_1, S_2, \ldots, S_i, \ldots, S_k$ during its execution. A sequence of 2-tuples $(L_1, T_1), (L_2, T_2), \ldots, (L_i, T_i), \ldots, (L_k, T_k)$ is associated with the sequence of states. In state S_i the program references the L_i locality set of pages for a duration of T_i. People who worked in the development of the LRU stack model assumed that a program has n locality sets at any time, n being the depth of the stack. The ℓth locality set consists of the ℓ most recently used pages, $1 \leq \ell \leq n$. "The true, or favored, locality set will then be the smallest set whose retention in memory leads to an acceptably low page fault rate" [SPIR76]. However, no method is provided to isolate one of the n localities as being the true locality set.

The work of Batson and Madison [BATS76a], [BATS76b], [BATS76c] is the only attempt found in literature to date to provide a formal definition of a locality set and a method to isolate locality sets in a reference string. To cure the deficiencies of the simple LRU model, Batson and Madison extended the LRU stack to include two new ordered vectors. Thus, at each moment of time t, three ordered vectors are kept to describe the state of the reference string:

$$P(t) = (P_1(t), P_2(t), \ldots, P_i(t), \ldots, P_n(t));$$

$$\sigma(t) = (\sigma_1(t), \sigma_2(t), \ldots, \sigma_i(t), \ldots, \sigma_n(t));$$

$$T(t) = (T_1(t), T_2(t), \ldots, T_i(t), \ldots, T_n(t)).$$

P(t) is the LRU stack of segment identifiers* as defined earlier. $\sigma_i(t)$ is the time at which the segment in the i-th stack position was last referenced. $T_i(t)$ is the time at which a reference was last made to a stack position greater than i. In other words, $T_i(t)$ is the time after which the i top positions of the stack were occupied by members of $S_i(t)$. $S_i(t)$ is the set of the i most recently referenced segments. At each time t, there is a hierarchy of sets $S(t) = (S_1(t), S_2(t), \ldots, S_i(t), \ldots, S_n(t))$. In this hierarchy $S_i(t) \subset S_{i+1}(t)$. $T_i(t)$ can be described as the formation time of $S_i(t)$. Figure 3 shows a reference string and its P(30), σ(30), and T(30) [BATS76a].

An <u>activity set</u> at time t is any set of segments in the LRU

*Batson and Madison studied only segmented virtual memory systems. We will discuss the implications of this limitation later.

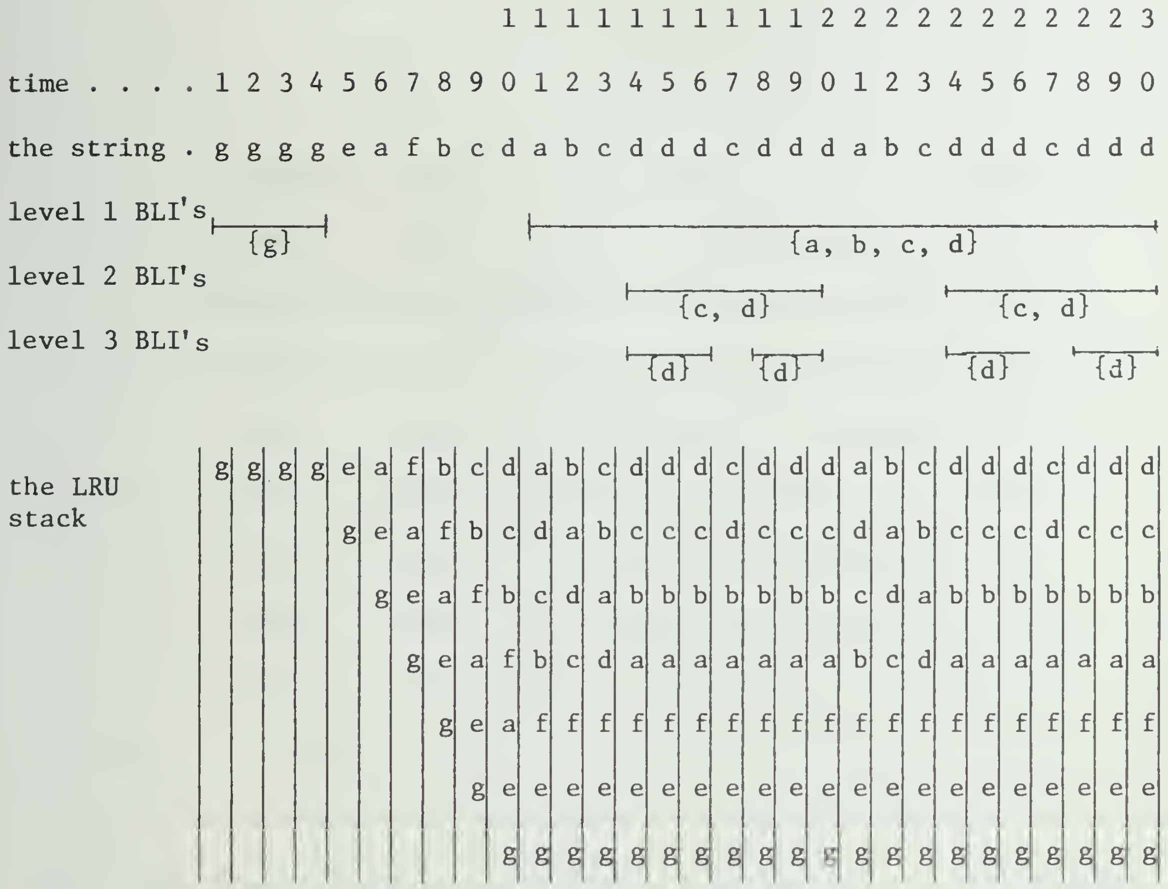

1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3
time 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
the string . g g g g e a f b c d a b c d d d c d d d a b c d d d c d d d
level 1 BLI's
{g}
{a, b, c, d}
level 2 BLI's
{c, d}
{c, d}
level 3 BLI's
{d}
{d}
{d}
{d}
the LRU stack
g g g g e a f b c d a b c d d d c d d d a b c d d d c d d d
g e a f b c d a b c c c d c c c d a b c c c d c c c
g e a f b c d a b b b b b b b c d a b b b b b b b
g e a f b c d a a a a a a a b c d a a a a a a a
g e a f f f f f f f f f f f f f f f f f f f f
g e e e e e e e e e e e e e e e e e e e e e
g g g g g g g g g g g g g g g g g g g g g

Figure 3-a. A Reference String, Its LRU Stack, and BLI's [BATS76a]

P(30)	S(30)	σ(30)	T(30)
d	$S_1(30) = \{d\}$	30	28
c	$S_2(30) = \{d,c\}$	27	24
b	$S_3(30) = \{d,c,b\}$	22	24
a	$S_4(30) = \{d,c,b,a\}$	21	11
f	$S_5(30) = \{d,c,b,a,f\}$	7	10
e	$S_6(30) = \{d,c,b,a,f,e\}$	5	10
g	$S_7(30) = \{d,c,b,a,f,e,g\}$	4	10

Figure 3-b. The P, S, σ, and T Vectors at $t = 30$ for the String in Figure 3-a [BATS76a]

hierarchy in which every member of that set has been re-referenced since the set was formed. In terms of the $\sigma(t)$ and $T(t)$ stacks, an activity set at time t, $A_i(t)$, is any $S_i(t)$ for which $\sigma_i(t) > T_i(t)$. At each instant during program execution, zero or more activity sets are recognized at various levels of the LRU hierarchy. Moreover, when a reference is made to any segment which is below a particular segment in the LRU stack, then this activity set (and any set above it) is terminated.

A <u>bounded locality interval</u> (BLI) is defined as the 2-tuple consisting of an activity set and its lifetime or residence at the top of the stack. In Figure 3, the BLI's of the example reference string are shown [BATS76a]. Notice the hierarchical structure of the BLI's. In [BATS76a] algorithms are given to update the $P(t)$, $\sigma(t)$, and $T(t)$ stacks. Also experimental results concerning the characteristics of the BLI's are presented in [BATS76a] and [BATS76c]. In Chapter 4 of this thesis, we will discuss Batson's experimental results and the validity of their implications. We have implemented Batson's algorithms and applied them to our collection of Fortran programs. We have correlated the syntactic structure of programs and found several problems with the concept of bounded locality intervals. Some of these are:

1. As is mentioned in [BATS76a], the way BLI's are defined lead to identifying a tremendous number of very short BLI's. These BLI's have no indication of locality or any significance. They only add undue expense to generating the experimental data. Figure 4 shows a real example taken from one of our programs. Only references

```
        DO  15  KK = 1,KMAX
        FD(KK) = FD(KK) + 273
        FE(KK) = FE(KK) + 1.E-3
       TTA(KK) = TTA(KK) + 273
15     QW1(KK) = QW1(KK) + 1.E-3
```

Figure 4-a. An Example Loop

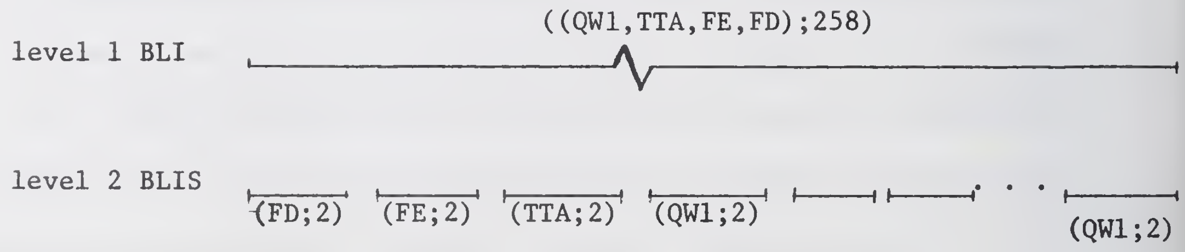


Figure 4-b. The BLI's Generated by the Program in Figure 4-a

to array elements are considered. Also every array is identified with only one segment. There is a one-to-one correspondence between array names and segment names. The level one BLI which is the true locality interval generated by the looping structure is of duration 258. However, every time each statement in the loop is executed, a level two BLI is generated with a duration of 2 references.

2. Long lived BLI's can be generated which will have a misleading indication of locality. Figure 5-a shows another real example from one of our programs which illustrates this situation. In Figure 5-b, we show the structure of the generated BLI's. In the first loop the arrays DZ, PO, QW1, TTA, RHO, FD, and FE are referenced. A level one BLI of duration 293 references and size 7 will be generated and it reflects a true locality interval because of the loop. In the following loop the referenced arrays are PI, QVS, HU1, FO, RHO1, QW1, PO, and TTA. In Figure 5-b, we notice that there are four BLI's covering the execution of the second loop. The BLI which is the true reflection of the second loop is the level 4 BLI. The BLI's at levels 1, 2, 3 are meaningless and give false indication of localities. Each of these contain some array names which are common to loops 1 and 3 and were never referenced in loop 2. Note that in Figure 4 the true BLI is the level 1 BLI but in Figure 5 the true BLI reflecting the second loop is the level 4 BLI. Thus there is no general rule which can be used to locate the BLI reflecting the real locality just by examining the BLI's generated from the trace of a program. We will elaborate on the confusion which the hierarchical structure of BLI's creates later.

```
          DO    1       I = 2,KMAX
          A1 = DZ(I)
          PO(I) = QW1(I) + TTA(I) + PO(I)
    1     RHO(I) = FD(I) + FE(I) + PO(I)

          DO    2       I = 1,KLES
          PO(I) = PO(I) + 5
          PI(I) = PO(I)/P
          FO(I) = PI(I)*2
          TTA(I) = TTA(I)/PI(I)
          QVS(I) = PO(I) * 3
          HU1(I) = QW1(I)/QVS(I)
          IF (HU1(I) . GE . .4) GO TO 3
          HU1(I) = . 4
          QW1(I) = QVS(I) * .4
      3   RHO1(I) = PO(I)/QW1(I)
    2     Continue

          DO     4     I = 2,KLES
          A1 = RHO(I)
          FD(I) = TTA(I) + 2
          FE(I) = QW1(I) + 1
          A2 = TTA(I) * 3
          BA(I) = RHO1(I) * TTA(I)/DZ(I)
          BB(I) = RHO1(I) + TTA(I)/(DZ(I)-5)
    4     Continue
```

Figure 5-a. An Example Program

```
level 1  ((PO,RHO,FE,FD,TTA,QW1,DZ);293) ((PO,PI,TTA,QVS,HU1,FO,RHO1,QW1,RHO,FD,FE,DZ);669)

level 2                                   ((PO,PI,TTA,QVS,HU1,FO,RHO1,QW1,RHO,FD,FE);666)

level 3                                   ((PO,PI,TTA,QVS,HU1,FO,RHO1,QW1,RHO);658)

level 4                                   ((PO,PI,TTA,QVS,HU1,FO,RHO1,QW1);655)
```

Figure 5-b. The BLI's Generated by the Example in Figure 5-a

The problem which is illustrated by the example in Figure 5 could be cured if the definition of an activity set was modified. If an activity set was defined as any set of segments of the LRU hierarchy in which every member of that set has been re-referenced k-times since that set was formed, $k > 1$, then we will have only one BLI covering the execution of loop 2 in the example of Figure 5. This modification will also reduce the number of very short BLI's. Although Batson in [BATS76b] mentions that Peter Denning did suggest this modification in the definition of an activity set to him, he did not modify the definition. The modification would increase the complexity and the expense of finding the BLI's in real traces of programs. Moreover, it is not obvious how one should choose k. The more important fact is that this suggestion does not really solve the problem of the confusion in interpreting the hierarchical structure of the BLI's. This is illustrated in the next example.

3. BLI's have an inconsistent correlation to the syntactic structure of programs. For example, the existence of a hierarchy of BLI's is a necessary but not sufficient condition for the existence of a nested loop in the source program. A nested loop will generate a multilevel hierarchical BLI structure. The existence of a multilevel BLI structure, however, can be due to other reasons. In Figure 6-a, the loop is double nested. This loop generates a two-level BLI structure. In the first loop of Figure 6-b, the arrays A, B, C, D, and E are referenced. A subset of this array set, namely, A, B, C, and D, are referenced in the

```
          DO    1  .I = 1,100
          A(I) = B(I) * C(I)
          DO    1  .J = 1,100
          D(I,J) = D(I,J) * A(I)
    1     Continue
```

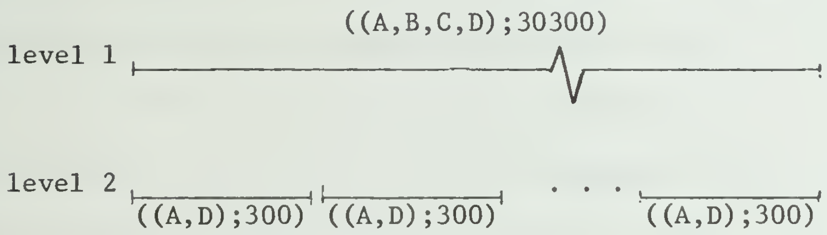


Figure 6-a. A Doubly Nested Loop and Its BLI's

```
      DO    10     I = 1,100
      A(I) = B(I) * C(I)
      D(I) = B(I) * E(I)
      C(I) = E(I) ** 2
10    Continue
      DO    20     I = 1,100
      B(I) = A(I) - C(I) * D(I)
20    Continue
      DO    30     I = 1,100
      E(I) = 0
30    Continue
```

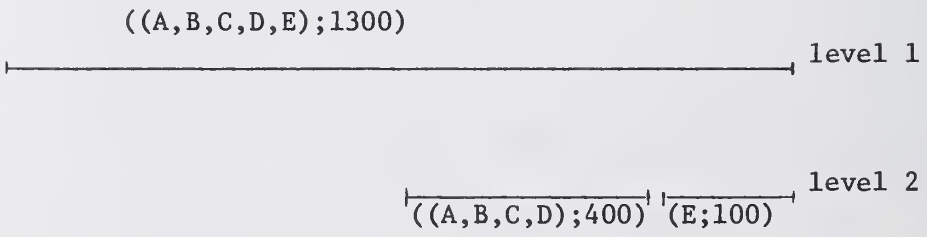


Figure 6-b. Consecutive Loops and their BLI's

second loop. The array E is referenced in the third consecutive loop. For this situation, we also have a hierarchical BLI structure. This structure is misleading. The first loop is not reflected in any BLI. The level one BLI hints at the existence of a locality of size 5 during the execution of the three loops. This is of course not true. In this situation, we really have three localities. The first one is of size 5, its members are A, B, C, D, E, and it covers only the first loop. This is followed by a locality of size 4, its members are A, B, C, D, and it covers the second loop. The last locality is of size 1, it contains the E array, and it covers the third loop. Denning's suggestion will not change the problem with the BLI's in this example.

4. There is no simple, obvious way of isolating the major phases of execution of a program from its BLI's. In other words, it is not obvious how to get the sequence of 2-tuples (L_1,T_1), (L_2,T_2), ..., (L_i,T_i), ... for a program from its BLI's.

In [BATS76a], level one BLI's of 10 milliseconds or greater duration are taken to be the major phases of execution. Our examples in Figures 5 and 6-b illustrate situations where level one BLI's give erroneous information. In Figure 6-a, the program spends most of its time referencing arrays of level 2 BLI's. To avoid these problems, a procedure is suggested in [BATS76b] to determine a pathway through the BLI hierarchy such that the space-time cost of executing the given program is minimized. The BLI's which are included in this pathway are taken to define the major phases of execution. The procedure suggested in [BATS76b] does not really minimize the space-time cost. The correct algorithms for minimizing the space-time cost of running a program were developed by Budzinski in [BUDZ77]. These algorithms are complex and expensive. Moreover, the localities of a program are supposed to be machine independent while in the approaches of [BATS76b] and [BUDZ77] the minimum space-time product is dependent on machine parameters such as the mean time needed to transfer a segment (or a page) from secondary to primary storage.

From the previous discussion, it is clear that the locality sets isolation problem has not really been solved. In the next section we present our different approach and solution to the problems presented in Sections 2.2.1 and 2.2.2.

2.2.3 Our Approach - Analysis of Program Behavior at the Symbolic Level

We think that there are two main reasons for the difficulties which people faced when trying to come up with satisfactory models of program behavior. The first reason is due to the approach taken in attacking this problem. Traditionally, people took reference strings generated by programs to be the observed phenomenon of interest. Thus, for them a program serves only the purpose of generating a reference string and then it can be ignored. However, the center of concern should really be the program itself and not the reference string. There is almost nothing important in a reference string which is not reflected in the source program. Thus, our approach will be to study and analyze programs at the source level. Although the complexity of programs was probably the main reason why people avoided studying programs at the source level, one can overcome this difficulty by recognizing that scientific programs have few basic structures. One can start by studying the most simple structure and then move to more elaborate ones. As it turns out, a clear understanding of simple structures can be extended rather easily to more complex ones. For more discussion about program analysis at the source level see [BATS76c].

The second reason for the difficulties of modeling program behavior is due to the programs themselves. As we will demonstrate later in this chapter, programs as written by people do not behave well in a paging environment.

We will adopt the following strategy in our study. First, we will develop a model for an ideal program. In developing this model we will discuss the important characteristics of such an ideal program. Next we show that it is possible to find some programs in the real world which follow this model. However, we will give examples of other programs which, as written by people, do not follow this model. In Chapter 3 we develop automatic transformation algorithms which can be used to force most programs to follow this model. Moreover, these transformations reduce the cost of execution of programs in virtual memory computers. Thus the transformations make programs behave better (they will be easier to model and manage) and cost less to execute.

In our study we will separate data and code pages. For programs with large data aggregates, code paging is trivial compared to data paging. We are mainly concerned with the data paging problem. Moreover, we will ignore references to scalars. These same assumptions were made in [BATS76a]-[BATS76c]. Most scientific production programs are written in Fortran. Moreover, there is a good reason to believe that versions of Fortran will continue to evolve and exist for a long time to come. Hence, without a loss of generality, we will use examples of Fortran-like programs and structures. All through this thesis we assume that paging will be made on demand. In other words we assume that there is no overlap between the CPU and I/O activities of the same program.

In Section 2.2.3.1 we develop our model of the program with the ideal behavior. In the same section we define elementary loops and show that such loops follow the model of the ideal program. Hence, we will call our model the elementary loop model (ELM). In Section 2.2.3.2 we will give examples of programs which do not follow the ELM model. In the same section we will mention those transformations of Chapter 3 which will cure specific problems with different programs.

2.2.3.1 The Elementary Loop Model

What is the ideal behavior of a program in a paged system? Ideally a program will need only a small fraction of its virtual space to be present in main memory. With this little memory allotment, the mean time between page faults, MTBPF, will be large. Moreover, the program will make effective use of the main memory page frames allotted to it. Thus, the density of reference to each page will be high. In other words the mean time between reference to each page, MTBR, will be small. Moreover, the page faulting activity will be clustered. This leads to rather long periods of useful CPU activity which are interrupt-free. This has an important effect in multiprogrammed systems. If programs have cyclic behavior in which they go through alternating periods of clustered I/O and CPU activities then the scheduling and other problems become much easier. The OS CPU time will be decreased.

The description given in the previous paragraph is that of a program which can be modeled by the ideal program model. Let us now define one kind of loops which follow this model.

Definition 1: An elementary loop is an ordered set of assignment statements preceded by one DO control statement. The variables referenced in the loop are one-dimensional arrays and possibly scalars. The subscripts of the array variables are linear functions of the index variable. In the subscript expressions, all the index variables have the same coefficient.

As an example of the behavior of an elementary loop let us discuss the behavior of the following program.

Program 1.

```
          DO     S3    I=1,N
      S1  A(I) = 2*I+3
      S2  C(I) = B(I)**2-4*C(I)
      S3  D(I) = C(I)/A(I)
```

Let Z be the number of words in a page, N>>Z, and $K = \lceil N/Z \rceil$. There are four arrays referenced in Program 1: A, B, C, and D. Each array occupies K pages of virtual space. Let us denote the ith page of A by a(i). Thus A will span the virtual pages a(1), a(2), ..., a(i), ..., a(K). Similar notation will be used for the pages of the other arrays. The total virtual space of these arrays is 4*K pages. In a non-virtual memory computer this program will need 4*K pages of main memory to run. If this amount of main memory is not available, the programmer must take care of transferring parts of his arrays between secondary and main memory such that the program will run in less than the total virtual space. In a virtual memory computer, however, the operating system will automatically take care of this problem. The operating system need only assign 4 pages to this program and the program will run in an optimum way under demand paging. It will have the minimum number of page faults, or I/O transfers between secondary and main memory. Moreover, its space-time cost will be minimum. With 4 pages of main memory, the program will have 4 page faults when it starts execution in order to allocate a(1), b(1), c(1), and d(1). After this burst of I/O activity the loop will go through Z iterations without any I/O interrupts. The I/O interrupt-free CPU activity will last for 7*Z memory references. 7 is the number of array memory references per iteration of the loop. During the CPU activity period the MTBR to the pages of the program will be $\leq$ 7 references. Thus the density of reference to these pages is high. Another burst or <u>cluster</u> of I/O activity will follow to allocate a(2), b(2), c(2), d(2) in main memory. In the next burst of CPU activity the loop index will go from Z + 1 to 2*Z and the duration of this CPU burst will be another 7*Z references. This oscillation or cycling between bursts of I/O and CPU activity will continue through the lifetime of this program. In the Ith cycle the pages a(I), b(I), c(I), and d(I) will be allocated and then processed. The I/O burst time will be 4*T, T being the average time of servicing a page fault (measured in memory references) and the duration of the CPU burst will be 7*Z references. The cycle time, T_c, will be 4*T + 7*Z references. Thus the mean time between the clusters of page faults is large, 4*T+7*Z. This behavior will be the same for the LRU, FIFO, or MIN replacement algorithms. The total number of page faults will be 4*K and the total space-time cost will be 4*K(4*T + 7*Z). This is a well behaved program. In a multiprogramming system, programs of this type will make the best use of the system. I/O and CPU bursts of different programs can be overlapped such that the I/O and CPU utilization will be maximized. The memory space will be saturated with different parts of different programs to maximize throughput. Such programs will run efficiently in virtual memory computers.

To have such a nice performance, Program 1 needs 4 pages of main memory. If 3 or less pages are assigned to it we will have one or more page faults per iteration. The number of page faults will be very large, O(N), instead of O(K), where N is the number of words in an array while K is the number of pages spanned by the array. In addition to the large increase in I/O activity, the program will lose the nice property of clustered page faults or bursts of I/O activity. The useful CPU activity will be constantly interrupted by page faults. The performance of the virtual memory system will collapse under such conditions.

For every elementary loop, there is a critical memory allotment which is needed in order to avoid performance collapse. In the case of Program 1 this number is 4. In general we will denote this number by m_o. The behavior of Program 1 and similar programs can be nicely modeled by the sequence of the 2-tuples:

$$(L_1, T_1), (L_2, T_2), \ldots, (L_i, T_i), \ldots, (L_k, T_k)$$

L_i = the ith locality set of pages

T_i = the residence time in this locality set of pages.

For Program 1 $L_i = \{a(i), b(i), c(i), d(i)\}$, and $T_i = 4{*}T + 7{*}Z$. The size of L_i, $|L_i|$, is equal to m_o which is constant at 4 for all i. Moreover, T_i is the same for all i and is equal to the cycle time, T_c, as discussed previously. Note that the phases of execution of this program have been easily identified.

Since the behavior of an elementary loop follows precisely the model of the ideal program, we will denote the ideal program model by the elementary loop model (ELM). Note that the ELM and an elementary loop are two different things. An elementary loop was defined in Definition 1. The ELM is the model of the ideal program. An elementary loop is an ideal program and it can be modeled using the ELM. Other loops, however, can also be modeled by the ELM. The following are the necessary conditions which must hold for a given loop so that it can be modeled by the ELM:

The Critical Memory Allotment,

m_o = 0 (# of different array names in the loop); 2.1

The Cycle Time,

$T_c = 0\ (R_\ell{*}c + m_o{*}T)$, where 2.2

R_ℓ = # of occurrances of array names in the loop,

c = integer constant (# of iterations per cycle);

Mean Virtual Time Between Clusters of

Page Faults, MTBPF = $0(R_\ell{*}c)$; 2.3

Mean Virtual Time Between References to a

page, MTBR = $0(R_\ell)$. 2.4

Equations 2.1-2.4 are the definition of the ELM.

Before proceeding any further, let us generalize an observation which we made concerning the execution of Program 1 to all elementary loops.

Theorem 1: Given an elementary loop L, let

m_o = the number of different array names referenced in the loop.

R_ℓ = the number of array references per iteration of the loop.

T = the average page fault service time.

K = the number of pages spanned by each array referenced in the loop.

With m_o page frames, the cost of executing the loop will be the same whether the replacement algorithm used is the LRU, FIFO, or Belady's MIN algorithm. The cycle time is given by:

$T_c = R_\ell * c + m_o * T$, where c = Z/(the coefficient of the index variable in the subscript expressions of the array variables).

The space-time cost is given by:

$$ST = T_c * m_o * K$$

Proof: When the execution of an elementary loop is started, m_o different pages will be referenced. If m_o page frames are allotted to the loop, all the three replacement algorithms will allocate these page frames to the first locality set of pages. In other words, the pages referenced in the first cycle of execution will be allocated space in main memory.

From our previous discussion in this section, the loop will have a cyclic behavior. We will use induction to prove our theorem. First, we show that the three replacement algorithms replace the set of pages referenced in the first cycle by those referenced in the second cycle. Second, given that the pages referenced in the (I-1)th cycle will be in memory when the Ith cycle is started, we will show that the three algorithms will replace these pages by the pages referenced in the Ith cycle.

When the first references to the pages of the second cycle are made, the MIN algorithm will replace pages referenced in the first cycle. This is because the forward distance of all these pages is infinite. Similarly, the LRU and FIFO algorithms will replace pages of the first cycle, though not necessarily in the same order. Thus, all these algorithms will produce m_o page faults and the second cycle time duration will be $R_\ell * c + m_o * T$. Note that c is the number of loop iterations per cycle.

If we assume that the pages of the (I-1)th cycle will be in memory when the execution of the Ith cycle starts then by a similar argument to the one presented in the previous paragraph, we conclude that the three algorithms will replace the pages of the (I-1)th cycle by those of the Ith cycle. Hence, in general the cycle time is given by $R_\ell * c + m_o * T$, and the total space-time cost is given by $(R_\ell * c + m_o * T)*m_o*K$.

Q.E.D.

The important point which Theorem 1 makes is that the performance of elementary loops will not be affected by the replacement algorithm used. It is totally determined by the amount of memory allotted. Note that Theorem 1 does not hold for the least frequently used replacement algorithm.

Although elementary loops are <u>not</u> a non-existing species in real programs, very often more complex loops will be encountered. Some of these can still be modeled by the ELM. Others, however, cannot. In the next section we will discuss some examples. In chapter 3 we will present two types of compile time optimizing transformations. The first type will be used to force any loop to behave such that it can be modeled using the ELM. The second type will be used to improve the cost of execution of loops; namely, to reduce the value of m_o, number of I/O transfers, and the space-time cost.

2.2.3.2 Other Loops

In this section we show examples of loops which are not elementary. Our examples fall in three categories. In the first category, the loops are not elementary but their behavior follows the ELM. Moreover, Theorem 1 holds for these loops. In the second category, the behavior of the loops follow the ELM model but Theorem 1 does not hold. The behavior of such loops is asymptotic to the behavior of elementary loops and they do not really have serious problems. In the third category the loops do not follow the ELM and their problems are serious. In Chapter 3 effort will be made to design transformations to cure the problems of such loops. A loop can be in one of these categories for different reasons. In what follows we give examples of these different reasons.

(i) Multi-dimensional arrays in loops.

The existence of large multi-dimensional arrays in a loop can easily cause problems in a virtual memory system. Let us first give an example in which multidimensional arrays cause no problem and the behavior of the program can still be modeled by the ELM. Consider the following loop:

```
Program 2.    DO    S1  I = 1,N
              DO    S1  J = 1,N
         S1   A(J,I) = B(J,I) + C(J,I)
```

In Fortran two-dimensional arrays are stored column-wise. In all our examples and analysis, we will consider large arrays which satisfy the condition $N \leq Z < N^2$. If each of the arrays of Program 2 spans K pages, then a close examination of the program will show that it can indeed be modeled by the ELM. For Program 2 the MTBR is $O(R_\ell)$ and

m_o = # different array names = 3

MTBPF = $T_c = Z * R_\ell + T * m_o \quad = Z * 3 + T * 3$

ST_c = space-time cost/cycle $\quad = 3 * (3 * Z + T * 3)$

The following program, however, cannot be modeled by the ELM:

```
Program 3.    DO   S1  I = 1,N
              DO   S1  J = 1,N
          S1  A(I,J) = B(I,J) * C(I,J)
```

To make the analysis simple let N = Z. We will make this assumption through the rest of this chapter. Each column of a matrix will span one page. The different number of array names here is still 3. With three page frames, however, three page faults will be generated per iteration of the inner-most loop. There is no clustering of page faults, i.e. CPU and I/O activities will be interleaved. Consequently, the system will suffer performance collapse. This loop needs all its virtual space to be allotted in main memory in order to generate the minimum number of page faults and to minimize its space-time cost.

The reason behind the difficulty with Program 3 is that the array elements are not being referenced in the order in which they were stored. If Program 3 was written in PL1, in which multi-dimensional arrays are stored row-wise, the problem would disappear. In PL1, however, Program 2 will have a problem. Thus it is obvious that for multi-dimensional arrays, the storage scheme and the pattern of reference are important in determining the behavior of a loop. This is what all of Elshoff's paper was about [ELSH74]; matching the pattern of reference to the storage scheme. In [McKE69] three storage schemes of multi-dimensional arrays were compared: row-wise, column-wise, and submatrix storage. If RZ = $\sqrt{Z}$, then in the submatrix storage scheme an (nxn) two-dimensional matrix will be divided into square submatrices of size

(RZ x RZ) as shown in Figure 7. If $N = \lceil n/RZ \rceil$ then there will be N^2 of these submatrices. Each submatrix is stored in a page. An m-dimensional array with the dimensions $D_1 \times D_2 \times D_3 \times \ldots \times D_m$ will be stored in $D_3 \times D_4 \times \ldots \times D_m$ planes. Each plane will contain $D_1 \times D_2$ array elements. There will be $\lceil D_1/RZ \rceil$ rows of pages and $\lceil D_2/RZ \rceil$ columns of pages in each plane. Hence each plane will have $\lceil D_1/RZ \rceil * \lceil D_2/RZ \rceil$ pages. The element of the array with the subscripts $d_1, d_2, \ldots, d_m$ will belong to the $\{(\lceil d_1/RZ \rceil - 1) * \lceil D_2/RZ \rceil + \lceil d_2/RZ \rceil + (d_3 - 1) * \lceil D_1/RZ \rceil * \lceil D_2/RZ \rceil + (d_4 - 1) * \lceil D_1/RZ \rceil * \lceil D_2/RZ \rceil * D_3 + \ldots + (d_m - 1) * \lceil D_1/RZ \rceil * \lceil D_2/RZ \rceil * D_3 * D_4 * \ldots * D_{m-1}\}$ page.

In [McKE69] it is shown that matrix algorithms can be designed such that with the submatrix storage scheme, enormous reduction in the number of page faults relative to row-wise storage can be achieved.

With 3 page frames and the submatrix storage scheme, Program 3 will have 3 page faults every RZ iterations of the inner-most loop. The duration of the interrupt-free CPU activity will be 3*RZ. This is not as good as the performance in Program 2 where the CPU burst time was 3*Z references long. Moreover, we still cannot use the ELM to model the behavior of program 3 even if the submatrix storage scheme is used to store the arrays. The problem here is that we will not reference all the elements involved in the calculation of each page while the page is in main memory. In Program 3 all the Z elements of a page will be referenced in the calculation while only RZ elements will be referenced every time the page is in main memory. Thus a given page will be transferred RZ times between secondary and main memory. In effect what

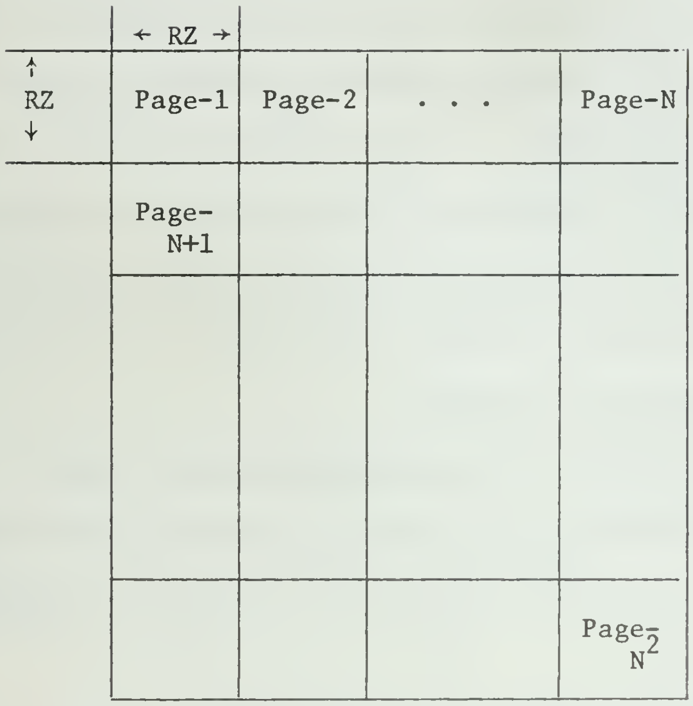


Figure 7. A Two-Dimensional Array Stored by the Submatrix Scheme

we are saying is that although the MTBPF for Program 3 is better with submatrix storage as compared with column storage (3*RZ compared to 3) it is still not as good as it is for Program 2 (3*Z).

In Chapter 3 the page indexing transformation will be introduced to cure the problems of multi-dimensional arrays. This is designed to transform a program such that all words of a page involved in a calculation will be referenced while the page is in main memory. We will adopt the submatrix storage scheme because of its inherent advantages as presented in [McKE69].

(ii) Mixing of arrays of different dimensions in a loop.

The performance of a loop can be affected in different ways when arrays of different dimensions are referenced. Consider the following example:

```
Program 4-a.  DO    3    J = 1,N
              DO    3    I = 1,N
              T(I,J) = .5 * DELT + TTA(I)
            3 continue
```

Since the elements of the two-dimensional array are referenced in the order in which they are stored, column-wise, the two-dimensional array represents no problem. This loop can be modeled by the ELM because equations 2.1 - 2.4 are satisfied. Namely, we have

$$m_o = O(\text{\# different array names}) = 2$$

$$\text{MTBR} = O(R_\ell) = 2$$

$$\text{MTBPF} = O(R_\ell * Z) = 2*Z$$

$$T_c = O(R_\ell * Z + m_o * T) = O(2*Z + 2*T)$$

Because of the existence of the one-dimensional array, T_c is not fixed through the execution of the program. In the first cycle two page faults will occur because t(1) and tta(1) must be allocated. Thus T_{c1} = 2*Z + 2*T. In the following cycles, however, only the t page will be replaced. Thus the steady state cycle time, T_{cs}, is given by 2*Z + T. Theorem 1 does not hold for this loop because the cycle time is not constant althrough the lifetime of the program.

In other situations, Theorem 1, will not hold for different reasons. For example, the following loop will not have identical performance under LRU and MIN replacement algorithms.

```
Program 4-b.  DO    3    J = 1,N
              DO    3    I = 1,N
              T(I,J) = T(I,J) + .5*TTA(J)
            3 continue
```

The reference string generated during the two iterations: (J = j-1, I = N) and (J = j, I = 1) is the following:

...,t(j-1), tta(1), t(j-1), t(j), tta(1), t(j),...

With 2 page frames under LRU, tta(1) will be replaced at the 4th reference to allocate t(j). MIN will replace t(j-1). Thus under LRU, T_{cs} = 3*Z + 2*T while under MIN T_{cs} = 3*Z + T. Note, however, that this loop can still be modeled by the ELM because equations 2.1 to 2.4 are satisfied.

In the previous two examples, mixing arrays of different dimensions in a loop did not present severe problems. Both loops could be modeled by the ELM although Theorem 1 does not hold for them. Their behavior is asymptotic to the behavior of elementary loops.

(iii) Loops with assignment statements at different nest levels.

Consider the following program:

```
Program 5.    DO      3       J = 1,N
              PT(J)  = TTA(J)
              DO      3       I = 1,N
              T(I,J) = .5* DELT + TTA(J)
         3    Continue
```

With 3 page frames, this loop will have $T_{cs} = (2*Z + 2) + T$ which is $O(R_\ell*Z + m_o*T)$. There is, however, an obvious waste in the space-time resource. The PT page is referenced only once during a cycle time. In other words, the N references made to PT are uniformly distributed through the execution time of the loop. This is reflected by the MTBR to the PT page which is O(N) instead of $O(R_\ell)$. Hence this loop cannot be modeled by the ELM. The loop distribution transformation presented in Chapter 3 will cure this problem.

As another example of a loop with large MTBR, consider the following program:

```
Program 6.    DO      10    I = 1,N
              A1 = W(I,1)* X(I,1)
              DO      10    J = 1,N
              A2 = WW * G(J,I)
               Y(J,I) = Y(J,I) + (A1-A2)/DZ
              A1 = A2
        10    Continue
```

Here, with 4 page frames, T_{cs} will be (3*Z + 2 + 2*T). The MTBR for the W and X pages is O(N) and not $O(R_\ell)$. Hence the ELM will not hold. A combination of the scalar expansion technique and loop distribution will handle the situation of this loop. This will also be discussed in Chapter 3.

(iv) IF statements in loops.

IF statements in loops will control the order of execution of assignment statements. Moreover they control which statements are to be executed during every iteration of the loop. Thus the memory requirement might in general vary between two cycles or even within one cycle. Moreover, the cycle time might vary from one cycle to another. Thus static measurements might not reflect an accurate estimate of the parameters of the ELM for a loop that contains an IF statement.

IF statements can be classified in several types [TOWL76]. One type of IFs called the A-type can be easily removed from the scope of the loop. The condition tested by a type-A IF is independent of the loop index and all variables computed within the loop. The result of the test will be the same for all iterations of the loop. This type of IF is illustrated in the following loop:

```
Program 7-a.    DO       10       I = 1,N
                IF (S.EQ.0)  GO TO 3
                A(I) = 4*B(I)*C(I) - D(I)**2
                GO  TO 10
            3   A(I) = 0
           10   Continue
```

The IF here is a static switch which can be removed as follows:

```
Program 7-b.        IF (S.EQ.0) GO TO 3
                    DO   101     I = 1,N
               101  A(I) = 4*B(I)*C(I) - D(I)**2
                    GO TO 103
                 3  DO   102     I = 1,N
               102  A(I) = 0
               103  Continue
```

One of the resulting two loops will be executed depending on the value of S. Each of the loops can be modeled by the ELM. It is important to note that we are not using the ELM to predict which parts of the program will be executed and which will not. What we are trying to do is to transform programs such that whatever loops are executed will be loops which can be modeled using the ELM.

In the other types of IFs, the condition tested will be a function of the index of the loop or some variables computed in the loop. These types of IFs cannot be removed outside the scope of the loop in the simple manner illustrated in Program 7. In many situations, however, the IFs do not affect all the statements within the loop. This is illustrated in the following two examples.

```
Program 8-a.        DO      S4   I = 1,N
               S1   B(I)  = G(I) - 7*DELT
               S2   IF(B(I) .GT.0)  C(I) = C(I)/B(I)*D(I)
               S3   C(I) = C(I) + 5
               S4   E(I) = D(I) * E(I)
```

Program 8-b.

```
        TEMP = 0
        DO    S3     I = 1,N
   S1   A(I) = B(I) * C(I) + 3
   S2   IF(TEMP. GT . A(I))  F(I) = 1
   S3   TEMP = TEMP + X(I) * Y(I)
```

In Program 8-a only S_2 is affected by the IF and in Program 8-b S1 is not affected by the IF. The loop distribution transformation will transform loops such that either the resulting loops are free of IF statements or all the assignment statements within the loop are affected by the IF statements such that they must be left in the same loop. In real programs, the number of statements and arrays in the latter type of loops is small and hence the variations in the parameters of the ELM for these loops are small.

2.3 Summary

In this chapter previous stochastic and deterministic models of program behavior were discussed. The difficulty of developing a simple accurate model of program behavior is due to the fact that programs as written by people are not well behaved from a paging system point of view.

The concept of the elementary loop model, ELM, was developed and the parameters of this model were discussed. Examples of programs which do not follow this model were presented. In the next chapter compiler transformations will be designed to cure such problems as those illustrated by the examples. Other transformations will aim at improving the ELM parameters of a given program. Thus, after applying the transformations of Chapter 3 to programs, they will be simple to model and cheap to run in a virtual memory computer.

3. PROGRAM TRANSFORMATIONS

A large portion of the early work in program analysis and transformations was motivated by the development of high speed parallel and vector machines like the ILLIAC IV, CDC STAR, and TI ASC around the turn of the decade. For these supercomputers, and the more recent ones, the Cray-1 and Burroughs Scientific Processor, the need for a vectorizing compiler is definite. The enormous computational power of these machines cannot be widely utilized by the general scientific community of users unless people can use ordinary high level languages to write programs for these machines. Moreover, there is an obvious need to be able to run the large amount of existing software, which was originally written for serial machines, on the new machines.

For the last few years research has been conducted at the University of Illinois to solve these problems. The problem of transforming ordinary serial programs to run on parallel and vector machines has been investigated and the results have been very good. A large software package called the PARAFRASE compiler evolved with the progress of these investigations. The PARAFRASE compiler takes an ordinary serial Fortran program and uses different compiler transformations to expose the inherent parallelism of the program [LEAS76],[WOLF78]. Pseudocode is generated and used to find the resulting speedup if the program were executed on parallel machines compared to serial machines.

The theme of this thesis is the enhancement of the performance of virtual memory computers. In this chapter we present program transformations to achieve this goal. These are intended to be optimizing compiler transformations which are tailored to cure the problems of large programs in virtual memory computers. Each transformation will serve one or both of two purposes. The first aim is to make programs follow the ELM and the second is to improve the parameters of this model for a given program. A transformation aimed at the first goal is a fix-up transformation. A transformation aimed at the second goal is an enhancement transformation.

Several of the concepts and transformations developed for speeding the execution of programs on parallel machines will be useful to us either with or without modifications. Thus we will use some of the transformations implemented in the PARAFRASE compiler, modify some, and introduce some new ones. We will think of the transformations and present them as source-to-source transformations. Our description of the transformations which were developed originally for parallel program execution will be very brief. We will present the modified and the new transformations in more details.

The flow chart shown in Figure 8 gives an overview of the general transformation process. This flow chart is intended to help the reader of this chapter understand the relationship between the different transformations and their relative order. Going back to examining this flow chart while reading this chapter will clarify the purpose and the logic behind the different transformations.

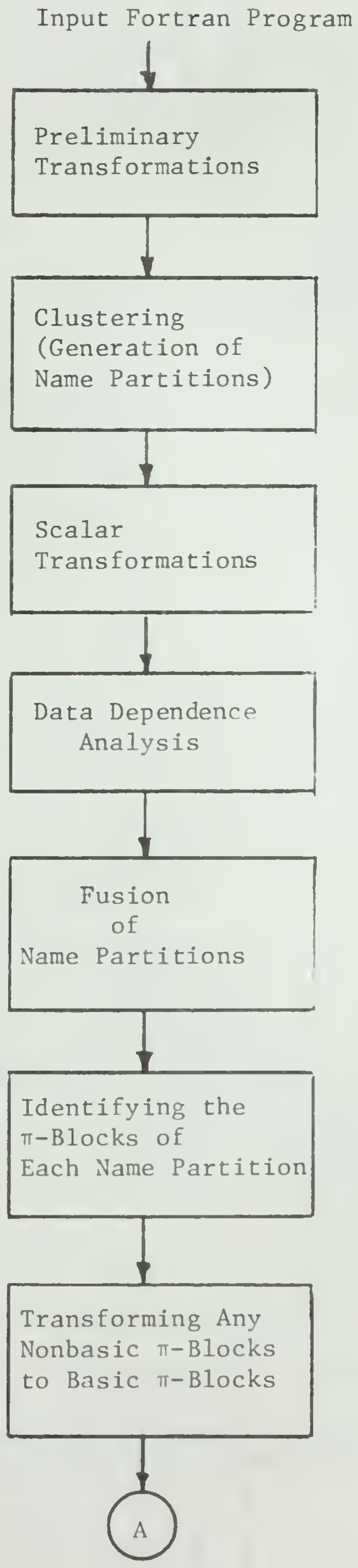

Input Fortran Program
Preliminary Transformations
Clustering (Generation of Name Partitions)
Scalar Transformations
Data Dependence Analysis
Fusion of Name Partitions
Identifying the π-Blocks of Each Name Partition
Transforming Any Nonbasic π-Blocks to Basic π-Blocks
A

Figure 8. An Outline of the Transformation Process.

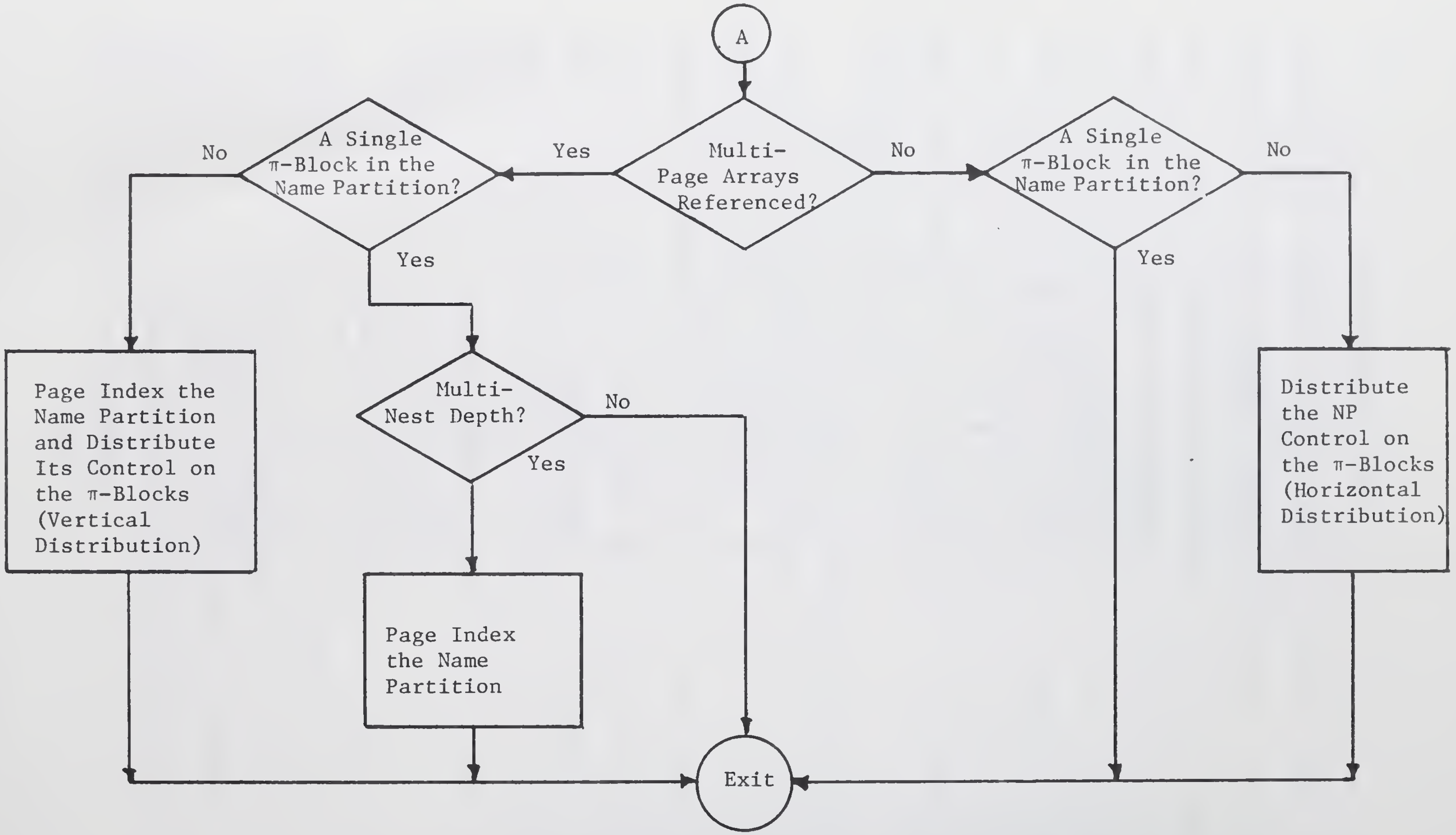

A
Multi-Page Arrays Referenced?
Yes
No
A Single π-Block in the Name Partition?
No
Yes
A Single π-Block in the Name Partition?
Yes
No
Page Index the Name Partition and Distribute Its Control on the π-Blocks (Vertical Distribution)
Multi-Nest Depth?
No
Yes
Page Index the Name Partition
Distribute the NP Control on the π-Blocks (Horizontal Distribution)
Exit

Figure 8 (continued). An Outline of the Transformation Process

In the preliminary transformations stage we apply (without modification) the following set of transformations which are currently implemented in PARAFRASE [WOLF78]:

(i) DO Loop Normalization.

(ii) IF Pattern Matching.

(iii) Scalar Renaming.

(iv) Induction Variable Substitution and Subscript Cleaning.

(v) Type-A IF Removal from DO Loops.

These transformations are aimed at breaking data dependences, and simplifying the control structure of the program. We will not discuss these transformations any more and refer the interested reader to [WOLF78].

Basic to the analysis of programs and development of transformations is the concept of data dependence. A brief discussion of this concept and related definitions will be presented in Section 3.1 and is based on [KUCK78],[TOWL76], and [BANE76].

In Sections 3.2 through 3.5 we discuss the rest of the transformations. In general we will present in each section some necessary definitions, some examples to illustrate the usefulness of the particular transformation, the transformation algorithm, and if needed some tests to check for the correctness of the transformation. We will try to strike a balance between formal and informal definitions of the transformations. A very formal definition leads to complex notations which explain unimportant details. Although we will present the transformations as separate entities, the intent is that all those relevant to a program segment will be applied.

3.1 Data Dependence Analysis

The set of input variables of an assignment statement S, IN(S), is the collection of variables appearing to the right of the assignment symbol. The output variable of S, OUT(S), is the variable which is assigned a value as a result of executing statement S. The output variable appears to the left of the assignment symbol. When S is executed each member of IN(S) is fetched from memory at least once and the output variable is stored in memory. Outside loops, an assignment statement S_q is said to be data dependent on another asignment statement S_p if $IN(S_q) \cap OUT(S_p) = x \neq \emptyset$ and the value computed in S_p for x is used in IN(S_q). We denote this by $S_p \Longrightarrow S_q$. If we have $x = OUT(S_p)$, $x \in IN(S_q)$, and the value of x computed in S_p is not used in S_q, then S_p is data antidependent on S_q. Antidependence is denoted by $S_q \not\Longrightarrow S_p$. S_q is said to be data output dependent on S_p, $S_p \overset{o}{\Longrightarrow} S_q$, if $x = OUT(S_p) = OUT(S_q)$ and the value calculated in S_q is stored in x after the value which is calculated in S_p. If $x = OUT(S_p)$, is a scalar variable then testing for dependence between S_p and statement S_q is simple and only involves name searching for x in IN(S_q) and OUT(S_q) and finding the order of execution of S_q relative to S_p. If x is an array element then the value of its subscripts in S_p and S_q should be identical in order for a dependence to exist.

The definition of dependence relations can be extended to cover statements in loops. Let us use $S_p[i_1, i_2, \ldots, i_d]$ to denote the instance of statement S_p during the particular iteration when $I_1 = i_1$, $I_2 = i_2$, $\ldots$, $I_d = i_d$. $I_1, I_2, \ldots, I_d$ are the index variables of the loop. Let

$x = OUT(S_p(k_1, k_2, \ldots, k_d))$. If we use the notation $S_p \underset{\circ}{<} S_q$ to denote that S_p is executed before S_q then we have [KUCK78]:

1) $S_p \Longrightarrow S_q$ if $x \in IN(S_q(\ell_1, \ell_2, \ldots, \ell_d))$ and $S_p(k_1, k_2, \ldots, k_d) \underset{\circ}{<} S_q(\ell_1, \ell_2, \ldots, \ell_d)$

2) $S_q \not\Longrightarrow S_p$ if $x \in IN(S_q(\ell_1, \ell_2, \ldots, \ell_d))$ and $S_q(\ell_1, \ell_2, \ldots, \ell_d) \underset{\circ}{<} S_p(k_1, k_2, \ldots, k_d)$

3) $S_p \overset{\theta}{\Longrightarrow} S_q$ if $x = OUT(S_q(\ell_1, \ell_2, \ldots, \ell_d))$ and $S_p(k_1, k_2, \ldots, k_d) \underset{\circ}{<} S_q(\ell_1, \ell_2, \ldots, \ell_d)$

Testing for dependence between statements within loops can be done by unrolling the loop and listing each statement for each iteration of the loop. Each statement can be checked with following statements for data dependence as described earlier. This testing procedure is lengthy and expensive. Tests for data dependence can be performed without actual unrolling of the loop. For array variables this involves testing the subscript expressions for the set of values which the index variable can take. In [BANE76] sufficient and necessary conditions for dependence are derived for index expressions that are linear functions of one index variable. For the rare case when the subscript expression is more complex or the subscripts are array elements, data dependence is usually assumed.

To simplify the testing procedures in [BANE76] it is assumed that the subscript expressions are functions only of the index variables. Moreover, the increment of an index variable between one iteration and the next is assumed to be 1. In [WOLF78] several transformations are described to ensure that index variables and subscript expressions satisfy these conditions.

In the previous discussion it was implicitly assumed that the loops are IF-free. In [TOWL76] procedures for removing IF statements from the scope of loops are described. Some types of IFs cannot be removed and in such situations it is currently assumed in the PARAFRASE compiler that all statements in the loop are interdependent. Research to improve the treatment of IFs is still going on.

The data dependence relations between statements in a block of assignment statements or a loop can be represented by a data dependence graph G. Each assignment statement S is represented by a node in the graph. If $S_i \Longrightarrow S_j$ we draw a directed arc of the type $\rightarrow$ from the node representing S_i to the S_j node. An arc of the form $\xrightarrow{\theta}$ is drawn from S_i to S_j if $S_i \overset{\theta}{\Longrightarrow} S_j$, and an arc of the type $\not\rightarrow$ is drawn from S_j to S_i if $S_j \not\Longrightarrow S_i$. Figure 9 shows a loop and its data dependence graph. Note the cycle in the graph. In general a cycle can exist in a graph if there are two statements S_p and S_q such that the relations $S_p \Delta S_q$ and $S_q \Delta S_p$ are both true. The relation $S_x \Delta S_y$ is defined by:

S_x (dependence operator) S_{i1} (dependence operator) ...(dependence operator) S_{in}(dependence operator) S_y, $n \geq 0$.

The dependence operator can be any of $\Longrightarrow$, $\not\Longrightarrow$, or $\overset{\theta}{\Longrightarrow}$. The Δ relation can be used to partition the nodes of a data dependence graph into a set of node partitions. Two nodes representing statements S_k and S_ℓ are in the same node partition, called a π-block, if and only if $S_k \Delta S_\ell$ and $S_\ell \Delta S_k$. In other words all the nodes which are in a cycle of the graph belong to the same π-block. A node which is not in a cycle is a π-block by itself. Later in this chapter an algorithm will be presented to distribute the loop control on its π-blocks.

	`DO 1 I = 2, 3`		
S_1	`A(I) = B(I-1)*3 + C(I)`	S_{11}	`A(2) = B(1)*3 + C(2)`
S_2	`C(I) = A(I+1)*3`	S_{21}	`C(2) = A(3)*3`
S_3	`B(I) = C(I)+A(I) + B(I)`	S_{31}	`B(2) = C(2)*A(2) + B(2)`
1	`CONTINUE`	S_{12}	`A(3) = B(2)*3 + C(3)`
		S_{22}	`C(3) = A(4)*3`
		S_{32}	`B(3) = C(3)*A(3) + B(3)`

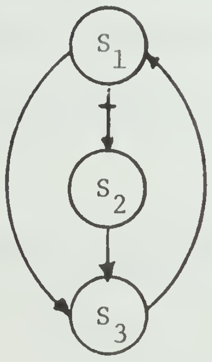


Figure 9. A Loop, Its Unrolled Version, and Its Data Dependence Graph

3.2 Clustering of Assignment Statements Algorithm

Programmers tend to group in the same loop different assignment statements which perform similar operations on different sets of arrays. Very obvious examples of such loops are initialization loops where different arrays of similar dimensions are initialized. This situation can also occur in loops where much more sophisticated calculations are performed. Examples of these loops are those performing similar calculations on real and imaginary parts of complex arrays.

The clustering transformation is designed to separate the set of statements inside a loop into several subsets such that in each subset a different group of arrays will be referenced. Each subset thus formed is called a name partition (NP). The transformation is applied to the loops of the program one at a time. The aim is to reduce the memory requirements of the program.

3.2.1 Definitions and Notations

Before describing the algorithm we make some definitions. For a particular loop L, let

$$S_L = (S_1, S_2, \ldots, S_i, \ldots, S_n)$$

be the ordered set of assignment statements controlled by L. For statement S_i, i is the ordering number. The set

$$A(L) = (a_1, a_2, \ldots, a_j, \ldots, a_m)$$

is the set of arrays referenced in L. If α is a subset of S_L, then the set of arrays referenced in α are denoted by $A(\alpha)$.

Definition 3.1 The name partitions of the loop L, are a set $(NP_1, NP_2, \ldots, NP_k)$ of subsets of SL with the following properties:

(i) $S_L = \bigcup_{q=1}^{k} NP_q$

(ii) $NP_i \cap NP_j = \phi$ for all $1 \leq i, j \leq k, i \neq j$

(iii) $A(NP_i) \cap A(NP_j) = \phi$ for all $1 \leq i, j \leq k, i \neq j$

(iv) $A(L) = \bigcup_{q=1}^{k} A(NP_q)$

(v) If $S_i \varepsilon NP_q$ and $S_j \varepsilon NP_\ell$ then there is no data dependence or data antidependence between S_i and S_j due to scalar variables.

3.2.2 The Clustering Algorithm

It is obvious from definition 3.1 that the control of a loop L can be distributed over its NP's. The order of execution of the resulting loops will be arbitrary. The NP's of a loop can be found by constructing an undirected clustering graph according to the following algorithm:

(i) Corresponding to each assignment statement draw a node and label it with the label of the statement.

(ii) For each array a_i referenced in the loop make a list, La_i, of the statements in which a_i is referenced.

(iii) Take every list formed in (ii) and travel through the nodes representing the statements in the list. When moving from one node to the next draw an undirected arc if no such arc existed because of a previous list.

(iv) Draw an arc, if one was not already drawn, between the nodes of any two assignment statements if there is a data dependence, or antidependence between the two statements due to a scalar variable.

(v) Divide the nodes of the graph into clusters. Each cluster will represent one NP and will contain the maximum number of connected nodes. Thus every pair of nodes in a cluster will be connected either directly or through other nodes which belong to the same cluster.

Figure 10 shows an example of applying the clustering algorithm to a loop. We note that the worst case complexity of the clustering algorithm is O(number of statements of the loop* number of variables referenced in the loop).

We now elaborate on the usefulness of the clustering transformation in reducing the cost of execution of multi-NP loops. If the original loop was assigned a number of page frames equal to its critical memory allotment, then one needs to assign to the transformed program only the maximum of the critical memory allotments of the resulting NP's. With this memory allotment the amount of I/O transfers will be the same for the original and transformed programs. Thus the space-time cost of the program will also be reduced by almost the same amount as its space requirement. This is true because the increase in the CPU time due to the additional control statements of the transformed program is not significant. One can establish a bound on the reduction of the space and the space-time cost. This is expressed in the following theorem:

Theorem 3.1 The upper bound on the improvement in the space requirement and the space-time cost of a loop due to the clustering transformation is a factor of K, where K is the number of name partitions generated by the clustering algorithm.

```
        DO  20  J = 1, NY1
        DO  10  I = 1, NX
S1      QVT1 = QV(I, J) + TS*QV1(I, J)     LQV = (S1, S3, S7, S9, S10)
S2      QCT1 = QC(I, J) + TS*QC1(I, J)     LQV1 = (S1, S3, S5)
S3      QV(I, J) = QV1(I, J)               LQC = (S2, S4, S8, S11, S12)
S4      QC(I, J) = QC1(I, J)               LQC1 = (S2, S4, S6)
S5      QV1(I, J) = QVT1
S6      QC1(I, J) = QCT1
10      CONTINUE
S7      QV(NX, J) = QV(1, J)            NP1 = (S1, S3, S5, S7, S9, S10);
S8      QC(NX, J) = QC(1, J)            NP2 = (S2, S4, S6, S8, S11, S12)
S9      QV(NXP, J) = QV(2, J)
S10     QV(NX+2,J) = QV(3, J)
S11     QC(NXP, J) = QC(2, J)
S12     QC(NX+2,J) = QC(3,J)
20      CONTINUE
```

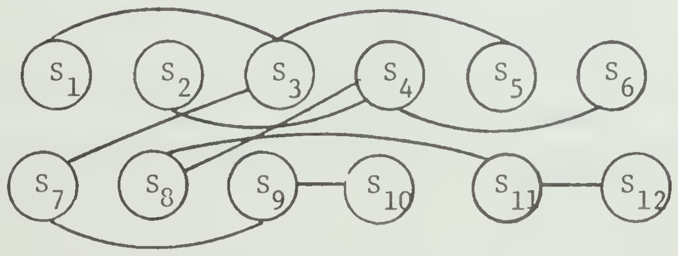


Figure 10. A Loop and Its Clustering Graph.

```
                     DO  201  J = 1, NY1
                     DO  101  I = 1, NX
        S_1          QVT1 = QV(I, J) + T_S*QV1(I ,J)
        S_3          QV(I, J) = QV1(I, J)
        S_5          QV1(I, J) = QVT_1
101                  CONTINUE
        S_7          QV(NX, J) = QV(1, J)
        S_9          QV(NXP, J) = QV(2, J)
        S_10         QV(NX+2, J) = QV(3, J)
201                  CONTINUE
                     DO  202  J = 1, NY1
                     DO  102  I = 1, NX
        S_2          QCT_1 = QC(I, J) + T_S*QC1(I, J)
        S_4          QC(I, J) = QC1(I, J)
        S_6          QC1(I, J) = QCT_1
102                  CONTINUE
        S_8          QC(NX, J) = QC(1, J)
        S_11         QC(NXP, J) = QC(2, J)
        S_12         QC(NX+2, J) = QC(3, J)
202                  CONTINUE
```

Figure 11. Distributing the Control of the Loop in Fig. 10 on Its NP's.

Proof:

The critical memory requirement of the original program, m_{oL}, is O(# of different array names in the loop). For the transformed program the critical memory requirement, m_o, is determined by the maximum of the number of array names in the different resulting NP's. If K is the number of NP's, then the smallest value which m_o can take is (m_{oL}/K). Since the clustering algorithm does not change the I/O time, the space-time cost will also drop by a factor of K.

Figure 11 shows the loop of Figure 10 with the control distributed on the NP's. Obviously the space and space-time cost are reduced by a factor of 2.

3.3 Fusion of Name Partitions

3.3.1 The Usefulness of the Fusion Transformation

The aim of this transformation is to reduce I/O time without increasing the memory requirements of a program. This is achieved by combining in one name partition several name partitions from different loops. The memory requirements of the combined name partition will not exceed the maximum memory needs of the individual NP's. As an example consider the following loops taken from a Fast Fourier Transform program:

Program 9-a.

```
            DO  6   K = K1, N2N, NDISP
            KPNG = K  + NG
     S1     CR(KPNG)  = CR(K)  - STOUTR(K)
     S2     CI(KPNG)  = CI(K)  - STOUTI(K)
```

```
          6    CONTINUE
               DO   8     K = K1, N2N, NDISP
          S_3  CR(K) = CR(K) + STOUTR(K)
          S_4  CI(K) = CI(K)+ STOUTI(K)
          6    CONTINUE
```

Using the clustering algorithm we get two NP's from the first loop: $NP_{11} = (S_1)$ and $NP_{12} = (S_2)$. We also get two NP's from the second loop: $NP_{21} = (S_3)$ and $NP_{22} = (S_4)$. If we distribute the loop control on the NP's we get the following program:

Program 9-b.

```
               DO   61   K = K1, N2N, NDISP
          61   CR(K + NG) = CR(K) - STOUTR(K)
               DO   62   K = K1, N2N, NDISP
          62   CI(K + NG) = CI(K) - STOUTI(K)
               DO = 81   K = K1, N2N, NDISP
          81   CR(K) = CR(K) + STOUTR(K)
               DO   82   K = K1, N2N, NDISP
          82   CI(K) = CI(K) + STOUTI(K)
```

The critical memory allotment of the first loop in the original program is four page frames and the total number of page faults is 4*K, K is the number of pages spanned by each array. The second loop has similar memory requirements and number of page faults. Thus the original program can execute in four page frames, the total number of page faults is 8*K, and the total space-time cost is 32*K. After applying the clustering transformation, Program 9-b needs only two page frames to execute without changing the number of page faults. Thus with clustering we have achieved an improvement of a factor of two in the memory requirement and space-time cost, without increasing the I/O time.

If we examine the arrays being referenced in the NP's, we find that $A(NP_{11}) = A(NP_{21}) = (CR,STOUTR)$ and $A(NP_{12}) = A(NP_{22}) = (CI,STOUTI)$. Moreover, the loop structure of NP_{11} and NP_{21} is identical; the nesting levels, the starting values of the index variables, the increment values, and the upper bound of the index set values, are all identical. We also notice that there are no data dependences between NP_{12} abd NP_{21}. Thus it is valid to combine NP_{11} and NP_{21} in one name partition, NP_1. Because of similar arguments we can combine NP_{12} and NP_{22} in a single name partition, NP_2. Hence after NP fusion the program will be transformed to the following:

Program 9-c.

```
      DO   1   K = K1, N2N, NDISP
      CR(K + NG) = CR(K) - STOUTR(K)
  1   CR(K) = CR(K) + STOUTR(K)
      DO   2   K = K1, N2N, NDISP
      CI(K + NG) = CI(K) - STOUTI(K)
  2   CI(K) = CI(K) + STOUTI(K)
```

The memory requirement of Program 9-c is the same as that of Program 9-b, namely, two page frames. Program 9-c, however, will produce less page faults: a total of 4*K page faults compared to 8*K page faults for the clustered and the original program. Table 4 compares the memory, I/O, and space-time cost of Programs 9-a, 9-b, and 9-c. We note that by

Table 4. Resource Requirements of Programs 9-a, 9-b, 9-c

	Critical Memory Allotment	Total Number of Page Faults	Space-Time Cost
Original Program	4	8*K	32*K
Clustered Program	2	8*K	16*K
Fusion Applied to the Clustered Program	2	4*K	8*K

using NP fusion of the clustered program we have improved the memory requirement, I/O, and space-time cost of the original program by factors of 2, 2, and 4 respectively.

3.3.2 Notation and Definitions

After illustrating the usefulness of the fusion transformation, we discussed some definitions relevant to the general fusion algorithm. The program is divided into a set of basic blocks. A basic block is defined as a section of code with only one point of entry and one point of exit. It contains a sequence of loops and possibly groups of assignment statements outside the loops. The fusion algorithm is applied to one basic block at a time. This is preceded by applying the clustering algorithm to the loops of the basic block. Let the number of loops in a basic block be m, $m \geq 1$. For loop L_k, $1 \leq k \leq m$, the clustering algorithm finds its set of n_k name partitions, $n_k \geq 1$. These are denoted by NP_{k1}, NP_{k2}, ..., NP_{kn_k}. The set of arrays references in NP_{ki} is denoted by $A(NP_{ki})$.

Although the NP's of one loop are by definition data independent, dependence relations can exist between NP's from different loops of a basic block of code. A name partition, NP_{ki} is data dependent on another name partition, NP_{qj}, $(k \neq q)$ if and only if there exists at least one statement in NP_{ki} which is data dependent on a statement in NP_{qj}. We denote this by $NP_{qj} \Longrightarrow NP_{ki}$. Similarly a data antidependence and data output dependence can exist between the name partitions NP_{ki} and NP_{qj} if and only if there exists at least one statement in NP_{ki} which is data antidependent or data output dependent on a statement in NP_{qj}. If NP_{ki} is data antidependent on NP_{qj} then this is denoted by $NP_{ki} \not\Longrightarrow NP_{qj}$. $NP_{qj} \Rightarrow\!\!\!\!\!\circ\; NP_{ki}$ means that NP_{ki} is data output dependent on NP_{qj}.

3.3.3 Correctness of Fusing Two Name Partitions

Before presenting the fusion procedure, let us discuss the question of the correctness of fusing two NP's, NP_{ki} and NP_{qj} $(k < q)$. When we fuse the two NP's we add the set of statements of NP_{qj} to those of NP_{ki}, i.e., $NP_{ki} = NP_{ki} \cup NP_{qj}$. The fusion of the two NP's will be valid if the following conditions are satisfied.

(A) The control structure of NP_{ki} and NP_{qj} is identical. This means that the index variable sets, and the nesting structure for the two NP's are identical.

(B) If $(NP_{\ell}, NP_{\ell+1}, \ldots, NP_{\ell+g})$ is the set of NP's between NP_{ki} and NP_{qj} then there is no data dependence, antidependence, or output dependence between NP_{qj} and any NP in this set. Moreover, there is no dependence between any assignment statement in NP_{qj} and any assignment statement which occurs outside NP's and between NP_{ki} and NP_{qj}.

We now present the general fusion algorithm. Again, we have m loops in a basic block of code (L_1, L_2, ..., L_m). Each loop L_k has n_k name partitions, (NP_{k1}, NP_{k2}, ..., NP_{kn_k}).

3.3.4 The Fusion Algorithm

(i) Set k = 1, ℓ = 1, i = 2

(ii) Compare $A(NP_{k\ell})$ with $A(NP_{i1})$. If $A(NP_{k\ell}) \subseteq A(NP_{i1})$ or $A(NP_{i1}) \subseteq A(NP_{k\ell})$ then test for the correctness of fusing $NP_{k\ell}$ and NP_{i1}. If the fusion can be done, replace $A(NP_{k\ell})$ by $A(NP_{k\ell}) \cup A(NP_{i1})$ and $NP_{k\ell}$ by $NP_{k\ell} \cup NP_{i1}$. Decrement n_i and eliminate NP_{i1} from the set of NP's of loop i. If n_i = o then decrement m.

(iii) Repeat step (ii) by considering $NP_{k\ell}$ and NP_{ij}, j = 2, ..., n_i.

(iv) If i = m go to step (v) else increment i and go to step (ii).

(v) If $\ell = n_k$ go to step (vi) else increment ℓ and go to step (ii).

(vi) If k = m exit, else increment k and go to step (ii).

We note that the complexity of this algorithm is $O((\text{total number of NP's in the basic code block})^2)$.

3.4 Scalar Transformations

Programmers usually introduce assignment statements with scalar output variables inside loops for different reasons. A scalar variable can be used as a temporary to hold the value of an expression which is common to several assignment statements. Sometimes the right-hand side expression of an assignment statement is very long and programmers prefer to divide the expression into parts to improve the readability of the program. Every part is assigned to a scalar variable and these are used in the right-hand side expression of the assignment statement. In another possibility the assignment statement to the scalar variable can be a recurrence.

3.4.1 The PARAFRASE Compiler Scalar Transformations

As will be shown in the next section, distributing the loop control of an NP on its π-blocks can be used to reduce the amount of memory required to execute the NP. In the PARAFRASE compiler several techniques are used to remove the arcs in the data dependence graph of a loop which are due to assignment statements to scalar variables. This will simplify the graph and reduce the number of statements included in a π-block. This is useful to us because, the smaller the number of statements in the π -blocks of an NP, the smaller the amount of memory which is needed for its execution. Of the techniques used in the PARAFRASE compiler to break data dependences due to scalars we use (without modification) the scalar renaming, induction variable substitution and forward substitution of right-hand sides of assignments statements to scalars which are used in subscript expressions. The dead code elimination pass will eliminate the assignment statements to those scalars treated by these techniques.

In the PARAFRASE compiler all scalars which cannot be handled by the previous three techniques will be expanded into array variables. Figure 12-a shows an example program and its data dependence graph. Notice the cycle in the graph. In Figure 12-b the scalar has been expanded into an array of size N and thus the cycle in the dependence graph has disappeared. The distribution algorithm which will be presented in the next section can be used to distribute the loop of the program in Figure 12-b. The program in 12-a is undistributable.

```
         DO  10  I = 1, N
S1       T = A(I) - E(I)
S2       A(I) = B(I)*C(I)
S3       B(I) = T + F(I)/D(I)
10       CONTINUE
```

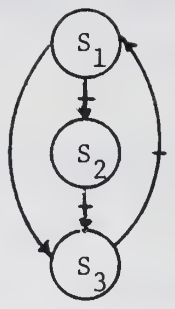


Figure 12-a. A Loop Including an Assignment Statement to a Scalar Variable and Its Data Dependence Graph.

```
         DO  10  I = 1, N
S1       T'(I) = A(I) - E(I)
S2       A(I) = B(I)*C(I)
S3       B(I) = T'(I) + F(I)/D(I)
10       CONTINUE
```

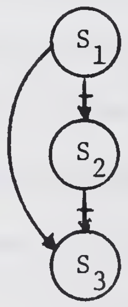


Figure 12-b. The Loop of Figure 12-a After Expanding the Scalar and Its New Data Dependence Graph.

3.4.2 The Scalar Forward Substitution Transformation

Figure 13-a shows another example program and its data dependence graph. The cycles in the graph are again due to an assignment statement to the scalar variable T. One can still use the scalar expansion technique to simplify the data dependence graph of the program in Figure 13-a. In fact, this is what is done in PARAFRASE. However, for this example the right-hand side of S_1 can be forward substituted in S_2 and S_3. S_1 can then be eliminated from the loop. This is shown in Figure 13-b. In PARAFRASE, this technique is not used because redundant computation might be introduced. This is the case in the loop of Figure 13-a because the scalar T is used in the right-hand side of two statements. Since PARAFRASE was written tc speedup program execution, forward substitution would not be a suitable transformation.

When people are compiling for parallel or pipelined machines they are not worried too much about the increase of the memory requirements of a transformed program if it can be executed on a parallel machine much faster than the original program on a serial machine. In this thesis we are concerned with compiler transformations for serial virtual memory computers. We are interested in a modified version of the PARAFRASE loop distribution transformation. In the next section we will describe our distribution transformation, the vertical distribution algorithm. Hence we are also interested in techniques to break data dependences in a loop which are introduced by assignment statements to scalar variables. However, we are concerned here with the memory requirement of the program and its I/O activity.

```
          DO  10  I = 1, N
S1        T = A(I)*C(I)
S2        D(I) = D(I)**2 - T**.5
S3        F(I) = T*(A(I) - C(I)) + F(I)/C(I)
10        CONTINUE
```

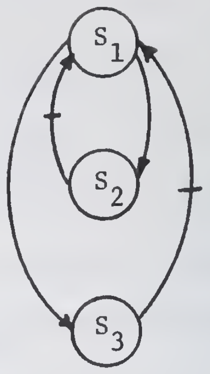


Figure 13-a. Another loop with an Assignment Statement to a Scalar Variable and Its Data Dependence Graph.

```
          DO  10  I = 1, N
S2        D(I) = D(I)**2 - (A(I)*C(I))**.5
S3        F(I) = (A(I)*C(I))*(A(I) - C(I))
                    + F(I)/C(I)
10        CONTINUE
```

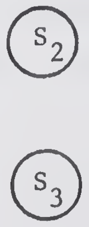


Figure 13-b. The Use of Forward Substitution to Simplify the Data Dependence Graph of Fig. 13-a.

Our approach will be to use the forward substitution technique in some situations and a modified version of the scalar expansion technique in other situations. Shortly we will give some rules to be used in deciding what to do for every specific case. Before presenting these rules we make one observation and then explain our modification to the scalar expansion transformation.

3.4.2.1 Correctness of the Forward Substitution Transformation

We note that the scalar expansion transformation can be applied to any scalar output variable of any assignment statement. This transformation is always correct as long as all references to the expanded scalar are replaced by references to appropriate elements of the resulting array. The details of the scalar expansion algorithm can be found in [WOLF78]. The forward substitution transformation on the other hand, cannot be used in all cases. For example it cannot be applied to the program in Figure 12-a. To address the correctness of the forward substitution transformation we make the following definitions.

Definition 3.2 If the output variable of an assignment statement is a scalar variable x, then this statement is called the source statement of the scalar x, S_x. The set of arrays referenced in S_x is denoted by AS_x.

Definition 3.3 A destination statement of a scalar variable x, D_x, is an assignment statement which is data dependent on S_x. In other words $S_x \Longrightarrow D_x$. We denote the set of array referenced in D_x by AD_x.

The necessary and sufficient condition for the correctness of the forward substitution transformation can now be stated as follows: If the source statement of a scalar variable x, S_x, is not a recurrence then its right-hand side expression can be forward substituted in a destination statement of x, D_x if and only if there is no statement executed after S_x and before D_x which is antidependent on S_x. If this condition is satisfied, then none of the input variables set of S_x changes its value before the execution of D_x and the substitution will be valid.

3.4.3 Modifying the Scalar Expansion Transformation

When a scalar variable is expanded into an array in the PARAFRASE compiler a different element of the array is used for every iteration of the loop. Thus for example, in Figure 12-b the expansion array, T', will be of size N. For execution on a parallel machine the loop can be distributed as shown in Figure 14-a. The distributed loop can be executed in 4 time steps on a parallel machine with N processor (we assume that performing any arithmetic operation takes one time step). On a serial machine, the program of Figure 12-a takes 4*N time steps to be executed. Thus a speedup of a factor of N has been achieved by distributing the loop.

In Section 3.5.1 we will show that although this kind of distribution, which we will call horizontal distribution, can result in reducing m_o of a loop, it may increase the I/O activity and possibly the space-time cost of the execution. In the same section we will modify the distribution algorithm to avoid any increase in the I/O activity and to ensure a reduction in the space-time cost. We will call our modified distribution algorithm, vertical distribution.

Figure 14-b shows the vertically distributed version of Program 12-a. By using vertical distribution the size of the expanded scalar need

```
          DO    S1  I = 1, N
S1        T'(I) = A(I) - E(I)
          DO    S2  I = 1, N
S2        A(I) = B(I)*C(I)
          DO    S3  I = 1, N
S3        B(I) = T'(I) + F(I)/D(I)
```

Figure 14-a. The Loop of Figure 12-b After Applying the Horizontal Distribition Transformation.

```
          DO  10  IP = 1, ⌈N/Z⌉
          ILB = 1 + (IP-1)*Z
          IUB = MIN(IP*Z, N)
          DO  S1  I = ILB, IUB
S1        T'(I MOD(Z) + 1) = A(I) - E(I)
          DO  S2  I = ILB, IUB
S2        A(I) = B(I)*C(I)
          DO  S3  I = ILB, IUB
S3        B(I) = T'(I MOD(Z) + 1) + F(I)/D(I)
10        CONTINUE
```

Figure 14-b. The Vertically Distributed Version of the Loop in Figure 12-b.

only be Z words, one page size. The expression (I MOD(Z)+1) is used as a subscript expression for the expanded array. Thus, with 4 page frames, the execution of the program in Figure 14-b starts by using the first page of A and the first page of E to compute one page of T' in S_1. In S_2 the first page of B and the first page of C are used to modify the first page of A. In S_3 the page of T' is used with the first page of F and the first page of D to write into the first page of B. In the second iteration of the outermost loop, the IP loop, the second pages of the arrays A, B, C, D, E, and F will be processed. However, the same page of T' can be again utilized to hold the temporary Z values computed in S_1 to be used in S_3. This will be true for all iterations of the IP loop.

Thus the difference between expanding scalars in parallel machine transformations and in virtual memory computer transformations is that the size of the expansion array in the latter case is less than or equal to one page size.

As mentioned before, the details of the expansion algorithm are found in [WOLF78]. We use the same algorithm except for reducing the size of the expansion array. In Figure 15, we show another example program and its vertically distributed version. Note that we have expanded the output scalar variable of statement S_1 into an appropriate one page array.

3.4.4 Choosing Between Scalar Expansion and Forward Substitution

When the control of an NP is vertically distributed on its π-blocks, the critical memory allotment, m_o, for each π-block will be roughly equal to the number of arrays referenced in it. Expanding the scalar output variable of an assignment statement into an array will increase m_o of the π-block containing this assignment statement by one

```
         DO  10  I = 1, N
         DO  10  J = 1, N
S1       T = A(I, J)
S2       A(I, J) = A(J, I) + B(I, J)
S3       A(J, I) = T + C(I, J)
10       CONTINUE
```

Figure 15-a. An Example Loop.

```
         DO  10  IP = 1, ⌈N/RZ⌉
         ILB = 1 + (IP-1)*RZ
         IUB = IP*RZ
         DO  10  JP = 1, ⌈N/RZ⌉
         JLB = 1 + (JP-1)*RZ
         JUB = JP*RZ
         DO  S1  I = ILB, IUB
         DO  S1  J = JLB, JUB
S1       T'(I MOD(RZ) + 1, J MOD(RZ) + 1) = A(I, J)
         DO  S2  I = ILB, IUB
         DO  S2  J = JLB, JUB
S2       A(I, J) = A(J, I) + B(I, J)
         DO  S3  I = ILB, IUB
         DO  S3  J = JLB, JUB
         A(J, I) = T'(I MOD(RZ) + 1, J MOD(RZ) + 1) + C(I, J)
10       CONTINUE
```

Figure 15-b. The Vertically Distributed Version of the Loop in Figure 15-a.

page frame. Moreover, all references to the scalar variable in other π-blocks must be replaced by references to the appropriate elements of the expansion array. Thus, m_o's for these π-blocks will also be increased. For example in Figure 14-b scalar expansion has increased the number of arrays referenced in both statements S_1 and S_3. However, by vertical distribution, which is possible in Figure 14-b because of scalar expansion, m_o is equal to 4 instead of 6 for the original loop in Figure 12-a.

If forward substitution is possible and if $AS_x \subseteq AD_x$ then substituting the right-hand side expression of S_x in D_x will not increase m_o of D_x. If this can be done for all the destination statement of x, S_x can be eliminated. Otherwise x must be expanded into an array and references to x in those statement for which $AS_x \not\subseteq AD_x$ must be replaced by references to appropriate elements of the expansion scalar.

From the previous discussion we conclude that scalar expansion should not be done unless it is incorrect to use the forward substitution transformation or if $AS_x \not\subseteq AD_x$ for some of the destination statements of x.

3.5 Distribution of Name Partitions

By applying the clustering and the fusion transformations to a program we expect to reduce its I/O activity, space requirement, and space-time cost. At this point of the transformation process, the different NP's of a basic block of code in the program will reference different sets of arrays. In a particular NP, however, it is not necessary that all the arrays of the NP will be referenced in each of its statements or even by any single statement. Thus it is intuitive that by distributing the control of an NP on its π-blocks its space requirements can be reduced to be roughly equal to the maximum number of arrays referenced in any of its π-blocks instead of the total number of arrays referenced in the NP.

In Section 3.5.1 we will present the distribution algorithm as currently implemented in the PARAFRASE compiler. In the same section we will differentiate between basic and nonbasic π-blocks. As mentioned previously, although this kind of distribution, the horizontal distribution, reduces m_o of an NP, it might increase its I/O activity and space-time cost. We will discuss an example to illustrate this point.

For NP's with basic π-blocks we describe the vertical distribution algorithm in Section 3.5.2. This is the horizontal distribution algorithm modified by the page indexing transformation. Vertical distribution reduces m_o of an NP but does not increase its I/O activity. In Section 3.5.2.1 we describe the algorithm when used for elementary loops. In Section 3.5.2.2 we discuss the algorithm when applied to multinested loops in which multi-dimensional arrays are referenced. In the same section we illustrate the use of the page indexing transformation in matching the pattern of reference of multi-dimensional arrays to their storage scheme. The general vertical distribution algorithm is presented in Section 3.5.2.3. Some implementation issues will also be considered in the same section. In Section 3.5.2.4 we present two theorems to be used in testing the correctness of applying the page indexing transformation.

Transforming NP's with nonbasic π-blocks is discussed in Section 3.5.3.

3.5.1 Horizontal Distribution of Name Partitions

We apply the horizontal distribution algorithm [KUCK78] to all NP's in which the set of arrays of the NP are not all referenced in every statement of the NP. If none of the arrays referenced in the NP is a multi-page array, horizontal distribution will be the last transformation applied to the NP. Otherwise, the method of distributing the control of the NP on its π-blocks will be modified using the page indexing transformation as will be described in the next section.

3.5.1.1 The Horizontal Distribution Algorithm

(i) By analyzing the subscript expressions and the index set for each index variable of the NP construct its data dependence graph.

(ii) Identify the π-blocks of the NP as defined in Section 3.1. We define the following partial ordering relations between two π-blocks, π_i and π_j:

(a) $\pi_i > \pi_j$ if and only if there exists $S_k \in \pi_i$ and $S_\ell \in \pi_j$ such that $S_k \Longrightarrow S_\ell$.

(b) $\pi_i \underset{\circ}{>} \pi_j$ if and only if there exists $S_k \in \pi_i$ and $S_\ell \in \pi_j$ such that $S_k \overset{\circ}{\Longrightarrow} S_\ell$.

(c) $\pi_j \overline{>} \pi_i$ if and only if there exists $S_k \in \pi_i$ and $S_\ell \in \pi_j$ such that $S_\ell \not\Longrightarrow S_k$.

We order the π-blocks of the NP according to these three relations. Note that the resulting ordering is not unique.

(iii) Distribute the NP control on its ordered π-blocks.

Figure 16 shows an NP, its data dependence graph, and its horizontally distributed version.

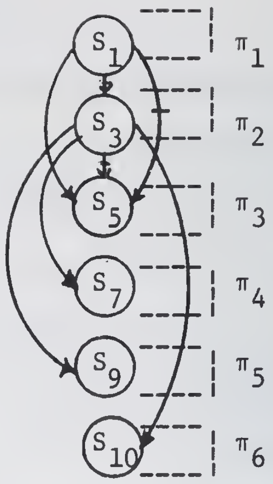
3.5.1.2 The Problem with Horizontally Distributing an NP with Multi-page Arrays

If multi-page arrays are referenced in different π-blocks of an NP, then the number of page transfers will be increased if the NP is horizontally distributed. As an example consider the program in Figure 17-a. The critical memory allotment for this NP, m_o, is equal to 3 and total number of page faults (using the LRU replacement algorithm) is $3*\lceil N/Z \rceil$. In the distributed NP of Figure 17-b, m_o is reduced to 2 page frames. However, the total number of page faults is increased to $6*\lceil N/Z \rceil$. The space-time cost is increased by a factor of $(2*6*\lceil N/Z \rceil)/(3*3*\lceil N/Z \rceil = 4/3$. In the undistributed NP, statements S_1, S_2, and S_3 will issue all their references to a particular page of the A array while this page is in main memory. In the horizontally distributed version, statement S_1 will issue its references to an A page,then this page will be replaced. The same page will be reloaded into main memory when it is referenced by S_2 and again when it is referenced by S_3. Similarly a B page will be loaded twice, once when it is referenced in S_2 and again when it is referenced in S_3. Note that the distributed program will have no problems if the size of each array was one page or less.

Before curing the increased I/O problem by adding the page indexing step to the horizontal distribution algorithm, let us differentiate between basic and nonbasic π-blocks.

```
          DO  S10  J = 1, NY1
          DO  S5   I = 1, NX
S1        QVT1'(I, J) = QV(I, J) + TS*QV1(I, J)
S3        QV(I, J) = QV1(I, J)
S5        QV1(I, J) = QVT1'(I, J)
S7        QV(NX, J) = QV(1, J)
S9        QV(NXP, J) = QV(2, J)
S10       QV(NX+2 , J) = QV(3, J)
```

(a) The NP



(b) The Dependence Graph

```
          DO  S1   J = 1,NY1
          DO  S1   I = 1, NX
S1        QVT1'(I, J) = QV(I, J) + TS*QV1(I, J)
          DO  S3   J = 1, NY1
          DO  S3   I = 1, NX
S3        QV(I, J) = QV1(I, J)
          DO  S5   K = 1, NY1
          DO  S5   I = 1, NX
S5        QV1(I, J) = QVT1'(I, J)
          DO  S7   J = 1, NY1
S7        QV(NX, J) = QV(1, J)
          DO  S9   J = 1, NY1
S9        QV(NXP, J) = QV(2, J)
          DO  S10  J = 1, NY1
S10       QV(NX+2, J) = QV(3, J)
```

(c) The Distributed NP

Figure 16. A Name Partition and Its Horizontally Distributed Version

```
          DO  S3  I = 1, N
S1        C(I) = C(I) - A(I)
S2        A(I) = 4*A(I)*B(I) - 2
S3        B(I) = B(I)*A(I) + B(I)
```

Figure 17-a. A Loop Referencing Multi-page Arrays

```
          DO  S1 I = 1, N
S1        C(I) = C(I) - A(I)
          DO  S2  I = 1, N
S2        A(I) = 4*A(I)*B*(I) - 2
          DO  S3  I = 1, N
S3        B(I) = B(I)*A(I) + B(I)
```

Figure 17-b. Horizontally Distributing the Loop of Figure 17-a

Definition 3.4 A basic π-block is a π-block in which all the statements are at the same nest depth level. Some of the statements of a nonbasic π-block will fall at different nest depth levels.

The vertical distribution transformation handles NP's which have only basic π-blocks. Such NP's are called basic NP's.

3.5.2 Page Indexing and Vertical Distribution of Basic Name Partitions

In the following subsections we will often need to refer to a set of consecutive integers. We now define a function, INT, which will denote such a set. We also give a formal definition of a basic NP.

Definition 3.5 Let w and k be two integers $w > 0$. The function INT(w,k) will denote the set of consecutive integers {(k-1)*w+1, (k-1)*w+2, ..., (k-1)*w+w-1, k*w).

Definition 3.6 A basic NP, BNP, is denoted by

$\text{BNP} = (I_1 \leftarrow \sigma_1, I_2 \leftarrow \sigma_2, \ldots, I_d \leftarrow \sigma_d)\ (B_1, B_2, \ldots, B_s)$ where I_j is an NP index, σ_j is an ordered index set, and B_j is a basic π-block or another BNP.

In some cases index variables of an NP are never used in subscript expressions of arrays. They are used as some kind of counters. We wish to differentiate between the DO statements associated with such index variables and those associated with index variables used in subscript expressions.

Definition 3.7 A type-A DO statement has an index variable which is used in some subscript expressions in an NP. If the index variable of a DO statement is never used in a subscript expression, then such a DO statement is of type B. In Figure 18, DJ and DI are type-A DO statements. DIJ is of type-B.

3.5.2.1 Vertical Distribution of Elementary NP's

By definition, an elementary NP has one DO statement and no multi-dimensional arrays. The NP in Figure 17-a is an example of elementary NP's. Let σ be the ordered index set. Let I_{min} be the smallest integer in σ and I_{max} be its maximum integer. If $I_{min} \in INT(Z,k_{min})$ and $I_{max} \in INT(Z,k_{max})$ then vertical distribution of the elementary NP means executing its first π-block, π_1, for the ordered index set $\sigma \cap INT(Z,k_{min})$, then executing π_2 for the same set and so on until the last π-block is executed for this set of values of the index variable. The same process is repeated for the ordered index set $\sigma \cap INT(Z,k_{min} + 1)$. We keep interating until we execute all the π-blocks for the last subset of the index variable set, namely $\sigma \cap INT(Z,k_{max})$.

Figure 19 shows the vertically distributed version of the NP in Figure 17-a. Vertical distribution is achieved by adding a set of statements called the page indexing statement set. In Figure 19 these are the ADD1I, ADD2I, and ADD3I statements. ADD1I is the paging DO statement. Its scope includes all the π-blocks in the NP. Statement ADD2I defines the lower bound of $\sigma \cap INT(Z,IP)$ for all the values of IP: 1, 2, ..., $\lceil N/Z \rceil$. Similarly ADD3I defines the upper bound of $\sigma \cap INT(Z,IP)$. We will refer to statements ADD2I and ADD3I as the lower bound and upper bound definition statements of the index variable I.

Note that the vertically distributed program in Figure 19 does not have the increased I/O problem of the horizontally distributed program in

```
DIJ        DO  10  IJ = 1, 3
           PK(1) = 1. - G*DZ/(2.*PT(1)*QVO(1))
DJ         DO  10  J = 1, NY1
           PK(J) = PK(J-1)*CP*QVO(J)
DI         DO  10  I = 1, NX1
           QV(I, J) = HUM(J)*QVS
           QV1(I, J) = QV(I, J)
10         CONTINUE
```

Figure 18. A Loop with Type-A and Type-B DO statements

```
ADD1I      DO  10  IP = 1, ⌈N/Z⌉
ADD2I      ILB = 1 + (IP-1)*Z
ADD3I      IUB = MIN(IP*Z, N)
           DO  S1  I = ILB, IUB
   S1      C(I) = C(I) - A(I)
           DO  S2  I = ILB, IUB
   S2      A(I) = 4*A(I)*B(I) - 2
           DO  S3  I = ILB, IUB
   S3      B(I) = B(I)*A(I) + B(I)
   10      CONTINUE
```

Figure 19. The Vertically Distributed Version of the NP in Figure 17-a

Figure 17-b. With two page frames, two page faults will occur when execution is started,to allocate two pages of the arrays A and C. This is followed by a burst of CPU activity during which the S_1 loop will be executed for Z iterations. A page fault will occur when a B page replaces the C page as the execution of the S_2 loop is started. The execution of this loop will last for 3*Z memory references. The same A and B pages will be used when the S_3 loop is executed for 4*Z references. Next the value of IP is incremented and a new execution cycle is started. Thus the number of page faults per cycle is 3 and the total number of page faults in $3*\lceil N/Z \rceil$. This is equal to the number of faults for the undistributed program. Since m_o was decreased from 3 to 2, the space-time cost was also decreased by the same factor, namely, 3/2. Table 5 compares the space, I/O time, and the space-time cost of the program in Figure 17-a, its horizontally distributed version, and its vertically distributed version.

3.5.2.2 Vertical Distribution of Multi-nested Basic Name Partitions with Multi-dimensional Arrays

As was mentioned in Chapter 2 we will adopt the submatrix storage scheme to store multi-dimensional arrays. We start this section by illustrating the usefulness of the page indexing transformation in matching the pattern of reference of multi-dimensional arrays to their storage scheme. We then describe using the page indexing transformation to vertically distribute multi-nested basic NP's which reference multi-dimensional arrays.

Consider the following matrix addition program

Table 5. Resource Requirement of the Program in Figure 17-a, Its Horizontally Distributed Version, and Vertically Distributed Version

	Critical Memory Allotment	Total Number of Page Faults	Space-Time Cost
Original Program	3	$3*\lceil N/Z \rceil$	$9*\lceil N/Z \rceil$
After Horizontal Distribution	2	$6*\lceil N/Z \rceil$	$12*\lceil N/Z \rceil$
After Vertical Distribution	2	$3*\lceil N/Z \rceil$	$6*\lceil N/Z \rceil$

Program 10-a

```
             DO 10 I = 1, N
             DO 10 J = 1, N
      10     A(I,J)  = B(J,I) + C(I,J)
```

Although the behavior of this program is improved by storing the arrays using the submatrix scheme rather than row-wise or column-wise, the MTBPF is still lower than predicted by the ELM. According to the ELM the MTBPF is 3*Z. For Program 10-a the MTBPF is 3*RZ, where $RZ = \sqrt{Z}$. The page indexing transformation will make the MTBPF equal to 3*Z. The transformed program is shown below.

Program 10-b.

```
  ADD1I        DO  10  IP = 1, ⌈N/RZ⌉
  ADD2I        ILB = 1 + (IP-1)*RZ
  ADD3I        IUB = MIN(IP*RZ,N)
  ADD1J        DO  10  JP = 1, ⌈N/RZ⌉
```

```
ADD2J     JLB = 1 + (JP-1)*RZ
ADD3J     JUB = MIN(JP*RZ,N)
          DO  10  I = ILB, IUB
          DO  10  J = JLB, JUB
10        A(I,J) = B(J,I) + C(J,I)
```

Again the idea here is to change the indexing pattern such that the maximum possible number of references are made to a page while the page is in primary memory. For this program we have two index variables I and J. The index variable set of I, $\sigma_I = (1, 2, \ldots, N)$, is divided into the subsets $(\sigma_I \cap \mathrm{INT}(RZ,1))$, $(\sigma_I \cap \mathrm{INT}(RZ,2))$, ..., $(\sigma_I \cap \mathrm{INT}(RZ,\lceil N/RZ \rceil))$. Similary σ_J is divided into similar subsets. The assignment statement in program 10-b will be first executed for the subsets $(I = \sigma_I \cap \mathrm{INT}(RZ,1)) \times (J = \sigma_J \cap \mathrm{INT}(RZ,1))$. Next the subsets $(I = \sigma_I \cap \mathrm{INT}(RZ,1)) \times (J = \sigma_J \cap \mathrm{INT}(RZ,2))$ will be used. The rest of the sequence of the index subsets will be: $\{(I = \sigma_I \cap \mathrm{INT}(RZ,1)) \times (J = \sigma_J \cap \mathrm{INT}(RZ,3))$; ...; $(I = \sigma_I \cap \mathrm{INT}(RZ,1)) \times (J = \sigma_J \cap \mathrm{INT}(RZ,\lceil N/RZ \rceil))$; $(I = \sigma_I \cap \mathrm{INT}(RZ,2)) \times (J = \sigma_J \cap \mathrm{INT}(RZ,1))$; ...; $(I = \sigma_I \cap \mathrm{INT}(RZ,\lceil N/RZ \rceil)) \times (J = \sigma_J \cap \mathrm{INT}(RZ,\lceil N/RZ \rceil))\}$. When IP = ip_1 and JP = jp_1 the index variables subsets will be $(I = \sigma_I \cap \mathrm{INT}(RZ,ip_1))$ and $(J = \sigma_J \cap \mathrm{INT}(RZ,jp_1))$. With this pattern of indexing, page indexing, all the elements of an A, B, or C page will be referenced before any elements of any other pages. Thus using page indexing and with 3 page frames, the program will have the minimum number of page fault, $3*\lceil N/RZ \rceil*\lceil N/RZ \rceil$. The MTBPF will be 3*Z and MTBR will be 3 references.

As is shown in Program 10-b page indexing was achieved by adding a paging statement set for every type-A DO statement in the program. For the DO statement of the I index variable, the statements ADD1I, ADD2I, and ADD3I were added. The scope of the ADD1I paging DO statement is identical to the scope of the I DO statement in the original program. The ADD2I defines the lower bound of the subset ($\sigma_I \cap$ INT(RZ,IP)) and the ADD3I statement define the upper bound of the same subset. Similar remarks apply to the ADD1J, ADD2J, and ADD3J statements.

Vertical distribution of multi-nested basic NP's can be achieved by adding a page indexing step to the horizontal distribution algorithm. After the π-blocks of the NP are identified, each type-A DO statement will be replaced by an appropriate paging statement set. Each π-block which was in the scope of a replaced DO statement will be enclosed by a new DO statement using the old index variable and the bounds of the index set as defined in the added paging statement set. The control of all type-B DO statements will be distributed on the relevant π-blocks without any modification. We illustrate the vertical distribution procedure by considering the following example:

Program 11-a.

```
DI        DO  10  I = 1, N
DJ        DO  10  J = 1, N
S1        C(I,J) = 0
DK        DO  10  K = 1, N
S2        C(I,J) = C(I,J) + A(I,K)*B(K,J)
10        CONTINUE
```

There are two π-blocks in this program $\pi_1 = \{S_1\}$, and $\pi_2 = \{S_2\}$. There are three type-A DO statements DI, DJ, and DK. The scope of DI and DJ includes both π_1 and π_2. The scope of DK includes only π_2. Thus the scope of the paging DO loops in the vertically distributed version of the program (ADD1I and ADD1J in the program below) will include π_1 and π_2. The scope of the paging loop in the statement set replacing DK will include only S_2. The vertically distributed version of Program 11-a is as follows:

Program 11-b.

```
ADD1I    DO  10  IP = 1, ⌈N/RZ⌉
ADD2I    ILB = 1 + (IP-1)*RZ
ADD3I    IUB = MIN(IP*RZ,N)
ADD1J    DO  10  JP = 1, ⌈N/RZ⌉
ADD2J    JLB = 1 + (JP-1)*RZ
ADD3J    JUB = MIN(JP*RZ,N)
         DO  S1  I = ILB, IUB
         DO  S1  J = JLB, JUB
S1       C(I,J) = 0
ADD1K    DO  10  KP=1, ⌈N/RZ⌉
ADD2K    KLB = 1 + (KP-1)*RZ
ADD3K    KUB = MIN(KP*RZ,N)
         DO  S2  I = ILB, IUB
         DO  S2  J = JLB, JUB
         DO  S2  K = KLB, KUB
S2       C(I,J) = C(I,J) + A(I,K)*B(K,J)
10       CONTINUE
```

Note that in this program a page of the C array will be initialized in π_1 then the same page will be referenced in π_2. Hence with vertical distribution,a page which is referenced in several π-blocks will not leave memory until it has been used in all these π-blocks.

3.5.2.3 Vertical Distribution of Basic NP's - the General Algorithm and Some Implementation Considerations

After introducing the concept of vertical distribution by examples in the previous two sections we now present the general algorithm.

(i) Construct the data dependence graph and identify the π-blocks of the NP as described in Section 3.5.1.1.

(ii) Start with the outmost type-A DO statement. Replace it by an appropriate page indexing statement set. The scope of the paging loop is the same as the scope of the replaced DO statement.

(iii) Enclose each π-block which was within the scope of the replaced DO statement by a DO statement using the same index variable. The upper and lower bounds of the index set are as defined in the added page indexing statement set. The increment is the same as in the replaced DO statement.

(iv) Repeat (ii) and (iii) for the next outermost type-A DO statement. This process continues until all type-A DO statements have been replaced. The control of all type-B DO statements will be distributed on the relevant π-blocks as done in the horizontal distribution algorithm.

We note that the added complexity of the distribution algorithm due to page indexing is 0 (# of DO statements in the NP).

In all the examples discussed in the previous sections all the subscript expressions were linear functions of one index variable, i.e., of the form a*index variable + β. Moreover, for these examples the coefficient of the index variable, a, was the same for all the subscript expressions and it was equal to 1. β was equal to zero in all expressions. If $\beta \neq 0$ for some expressions, we will still use the same implementation techniques as illustrated in the examples. If $a \neq 1$ but it was the same number, c, for all subscript expressions, our implementation method can be modified slightly to accomodate such cases. This is illustrated in the following example.

Program 12-a.

```
              DO   1   I = 1, N
     S1       A(3I) = B(3I)*3
     S2       D(3I) = B(3I-1)/3
     1        CONTINUE
```

The vertically distributed version of this program is shown below

Program 12-b.

```
     ADD1I    DO   1  IP = 1,⌈N/⌊Z/3⌋⌉
     ADD2I    ILB = 1 + (IP-1)*⌊Z/3⌋
     ADD3I    IUB = MIN(IP*⌊Z/3⌋,N)
              DO   S1  I = ILB, IUB
     S1       A(3I) = B(3I)*3
              DO   S2  I = ILB, IUB
     S2       D(3I) = B(3I-1)/3
     1        CONTINUE
```

We note that $\lfloor Z/3 \rfloor$ is the number iterations which is spent by Program 12-a referencing one page of A, one page of B, and one page of D. Thus $\lceil N/\lfloor Z/3 \rfloor \rceil$ is total number of pages of A referenced. Similarly the same number of pages of the B and C arrays are referenced. Program 12-b will have $\lceil N/\lfloor Z/3\rfloor\rceil$ cycles. In each cycle $2*\lfloor Z/3\rfloor$ references will be made to two pages of A and B in the S_1 loop. This is followed by $2*\lfloor Z/3\rfloor$ references made to the same B page and a D page in the S_2 loop.

In general, if the coefficient of all the index variables in all the subscript expressions was the number c, Z should be replaced by $\lfloor Z/c\rfloor$ in the added statement set (or RZ should be replaced by $\lfloor RZ/c\rfloor$ when multi-dimensional arrays are involved). If the coefficient of the index variables were not the same for all subscript expressions, we use their minimum, c_{min}. Thus Z will be replaced by $\lfloor Z/c_{min}\rfloor$ in the added statement set. Such cases ,where the subscript expressions are more complex functions of one or more index variables ,are of little practical interest and hence we will not discuss such cases any further.

Before leaving this section we remark that the lower bound of the added paging DO statement was equal to 1 in all our examples. This is not true in the general case. If the lower bound of the index set of the replaced DO statement was I_{min} and $I_{min}\varepsilon INT(Z,k_{min})$ then the lower bound of the paging index set will be k_{min} (in the case where multi-dimensional arrays are involved $I_{min}\in INT(RZ,k_{min})$). Note that the added lower bound definition statement should be adjusted to make ILB = I_{min} when IP = k_{min}.

3.5.2.4 <u>The Correctness of the Page Indexing Transformation</u>

The correctness of the horizontal distribution algorithm is obvious from the definition of data dependences and π-blocks. When page indexing is used to achieve vertical distribution, the order of referencing elements of multi-dimensional arrays in π-blocks is different from the order of their reference as specified in the undistributed program. Thus we need to establish some necessary and sufficient conditions which can be used to test whether the page indexing transformation is valid. We will illustrate the problem by considering the following example.

Program 13-a.

```
                DO   1   I = 1, 48
                DO   1   J = 1, 48
    S1          A(I,J) = B(I,J)*2
    S2          C(I,J) = A(I-1, J+1)/2
    1           CONTINUE
```

In this program there is one dependence relation, namely S_2 is data dependent on S_1. Thus there will be no cycles in the data dependence graph and the program can be horizontally distributed as shown below.

Program 13-b.

```
                DO  S1  I = 1, 48
                DO  S1  J = 1, 48
    S1          A(I,J) = B(I,J)*2
                DO  S2  I = 1, 48
                DO  S2  J = 1, 48
    S2          C(I,J) = A(I-1, J+1)/2
```

For a page size of 64 words we get the following program if we apply page indexing to Program 13-b.

Program 13-c

```
    ADD1I       DO  10  IP = 1, 6
    ADD2I       ILB = 1 + (IP-1)*8
    ADD3I       IUB = IP*8
```

```
ADD1J       DO  10  JP = 1, 6
ADD2J       JLB = 1 + (JP-1)*8
ADD3J       JUB = JP*8
            DO  $S_1$ I = ILB, IUB
            DO  $S_1$ J = JLB, JUB
$S_1$          A(I,J) = B(I,J)*2
            DO  $S_2$  I = ILB, IUB
            DO  $S_2$  J = JLB, JUB
$S_2$          C(I,J) = A(I-1, J+1)/2
```

Program 13-c will produce erroneous results. To see this consider for example the value assigned to C(2, 8) in S_2. On the right-hand side of S_2 the value of A(1, 9) is used in computing C(2, 8). In Programs 13-a and 13-b this value of A(1, 9) will be computed in S_1. In Program 13-c the value of A(1, 9) used to compute C(2, 8) is an old value, i.e., when the assignment to C(2, 8) is made the new value computed in S_1 for A(1, 9) has not been stored in A(1, 9) yet. Hence Program 13-a cannot be vertically distributed.

To simplify our discussion of this subject we will consider only a basic π-block with only one assignment statement. This will be of the form:

Program 14-a.

```
            DO   S   $I_1$ = 1, N
            DO   S   $I_1$ = 1, N
S           $A(F_1(I_1), F_2(I_2)) = A(f_1(I_1), f_2(I_2))$ + < an expres-
                                 sion not containing references
                                 to A>
```

where $F_1(I_1)$ and $f_1(I_1)$ are linear functions of I_1. Similarly $F_2(I_2)$ and $f_2(I_2)$ are linear functions of I_2. At the end of this section we will discuss extending our analysis and theorems to cover more general cases.

The problem here is to find sufficient and necessary conditions for the correctness of page indexing Program 14-a, i.e., we want to test whether the following program will produce identical results to those produced by Program 14-a:

Porgram 14-b.

```
              DO   S   $IP_1$ = 1, ⌈N/RZ⌉
              $ILB_1$ = 1 + ($IP_1$-1)*RZ
              $IUB_1$ = MIN($IP_1$*RZ,N)
              DO   S   $IP_2$ = 1, ⌈N/RZ⌉
              $ILB_2$ = 1 + ($IP_2$-1)*RZ
              $IUB_2$ = MIN($IP_2$*RZ, N)
              DO   S   $I_1$ = $ILB_1$, $IUB_1$
              DO   S   $I_2$ = $ILB_2$, $IUB_2$
      S       A($F_1(I_1)$, $F_2(I_2)$) = A($f_1(I_1)$, $f_2(I_2)$) + ...
```

Figure 20-a shows the $I_1 x I_2$ plane. Each point (i_1, i_2) in this plane can be associated with the execution of the statement S when $I_1 = i_1$ and $I_2 = i_2$. One can imagine a cursor that moves from one point to another in the $I_1 x I_2$ plane as S is executed with the index variables taking the values of the coordinates of the first point,then executed with the index variables taking the values of the coordinates of the second point, etc. Thus the cursor will trace a particular curve in the $I_1 x I_2$ plane during the execution of S(actually it will visit discrete

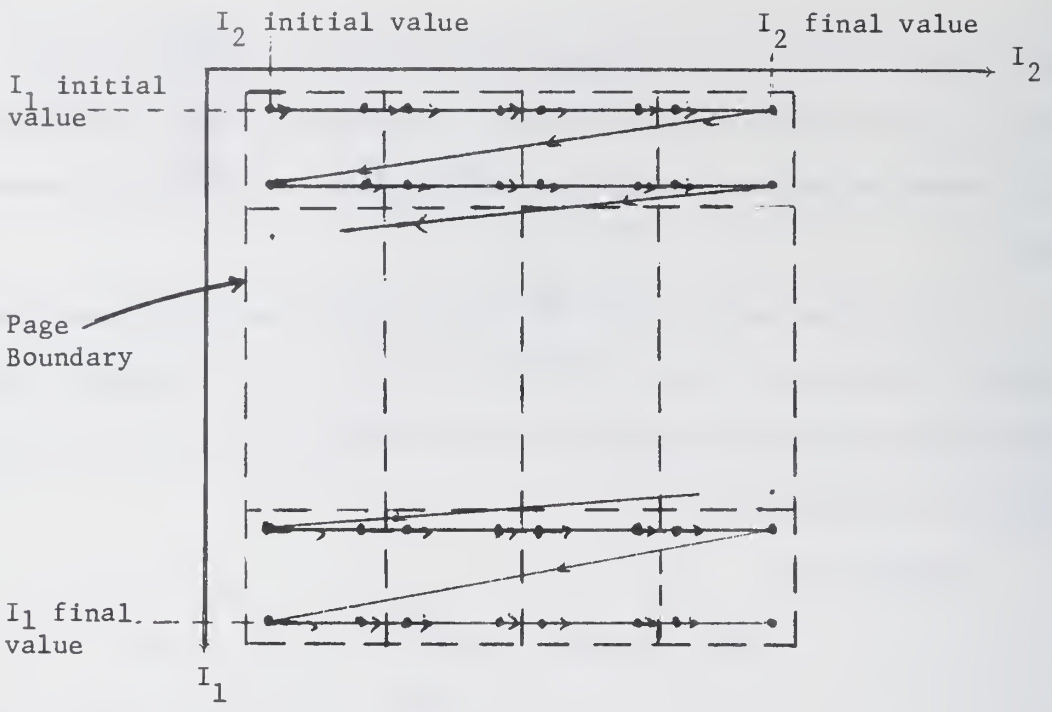


Figure 20-a. The Curve Traced by the Cursor in the $I_1 x I_2$ Plane when Program 14-a Is Executed.

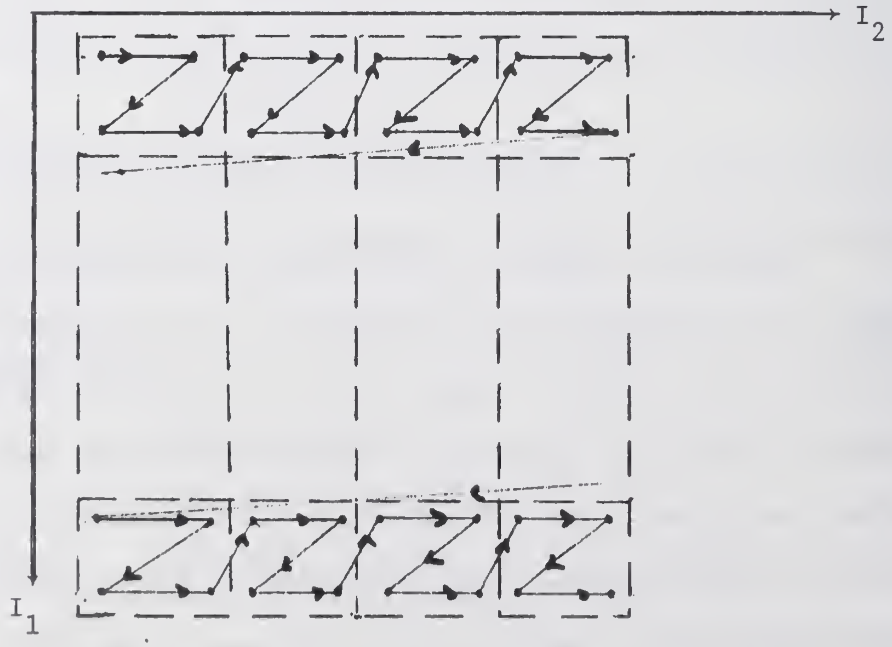


Figure 20-b. The Curve Traced In the $I_1 x I_2$ Plane when Program 14-b Is Executed.

points on the curve). Figure 20-a shows the curve traced by the cursor when Program 14-a is executed. In the figure N = 8.

If the cursor passes through the point P with the coordinates (i_1, i_2) before the point P' with the coordinates (i'_1, i'_2) we will say that P precedes P' and denote this by $P < P'$ or $(i_1, i_2) < (i'_1, i'_2)$.

Figure 20-b shows the curve traced by the cursor when Program 14-b is executed. In the figure RZ = 2.

According to the execution sequencing of S in Program 14-a, if $OUT(S(i'_1, i'_2)) \varepsilon IN(S(i_1, i_2))$ and $(i'_1, i'_2) < (i_1, i_2)$ then there is a dependence vector, $V = (i_1, i_2)(i'_1, i'_2)$ from point P' to point P in the $I_1 x I_2$ plane. In general, if there are references to several different elements of A on the right-hand side of S, there might be several dependence vectors from several points, P', P", P'", ... to the point P. The points, P', P", ... are called the source points of these dependence vectors and the point P is the destination point. The page indexing transformation will be correct if and only if, for all computed points, the cursor will pass through all the dependence source points of each given point before it passes through the point itself.

As an example consider the program:

Program 15 .

```
                DO  10  I1 = 1, 4
                DO  10  I2 = 1, 4
   10           A(I1 + 1, I2) = A(5-I1, 5-I2)
```

Table 6 lists the points visited by the cursor, their coordinates in the $I_1 x I_2$ plane, $OUT(S(I_1,I_2))$, and $IN(S(I_1, I_2))$. Examining the table we find the following dependence vectors:

Table 6. The Points on the Execution Trace of Program 15.

Point	The Coordinates i_1	i_2	$OUT(S(i_1, i_2))$	$IN(S(i_1, i_2)$
P_1	1	1	a(2, 1)	a(4, 4)
P_2	1	2	a(2, 2)	a(4, 3)
P_3	1	3	a(2, 3)	a(4, 2)
P_4	1	4	a(2, 4)	a(4, 1)
P_5	2	1	a(3, 1)	a(3, 4)
P_6	2	2	a(3, 2)	a(3, 3)
P_7	2	3	a(3, 3)	a(3, 2)
P_8	2	4	a(3, 4)	a(3, 1)
P_9	3	1	a(4, 1)	a(2, 4)
P_{10}	3	2	a(4, 2)	a(2, 3)
P_{11}	3	3	a(4, 3)	a(2, 2)
P_{12}	3	4	a(4, 4)	a(2, 1)
P_{13}	4	1	a(5, 1)	a(1, 4)
P_{14}	4	2	s(5, 2)	a(1, 3)
P_{15}	4	3	a(5, 3)	a(1, 2)
P_{16}	4	4	a(5, 4)	a(1, 1)

$V_{12} = \{(3,4); (1,1)\}$

$V_{11} = \{(3,3); (1,2)\}$

$V_{10} = \{(3,2); (1,3)\}$

$V_9 = \{(3,1); (1,4)\}$

$V_8 = \{(2,4); (2,1)\}$

$V_7 = \{(2,3); (2,2)\}$

Figure 21 shows these dependence vectors in the $I_1 x I_2$ plane. If this program is page indexed for a page size of 4 the cursor will visit the points of the $I_1 x I_2$ plane in the following order:

$$P_1, P_2, P_5, P_6, P_3, P_4, P_7, P_8, P_9, P_{10}, P_{13}, P_{14}, P_{11}, P_{12}, P_{15}, P_{16}$$

We note that the source point of every dependence vector is visited before its destination point. Thus the page indexing transformation is valid for a page size of 4. The transformation will not be valid, however, for a page size of 9. In this case P_9 will be visited before P_4.

We present next a theorem to be used in testing the validity of the page indexing transformation for all page sizes.

<u>Theorem 3.2</u> For the program:

```
            DO  S  $I_1 = 1, N$
            DO  S  $I_2 = 1, N$
    S       $A(F_1(I_1), F_2(I_2)) = A(f_1(I_1), f_2(I_2))$ + < an
                                   expression not containing
                                   references to A >
```

Let F_1, f_1 be linear functions of I_1 and F_2, f_2 be linear functions of I_2.

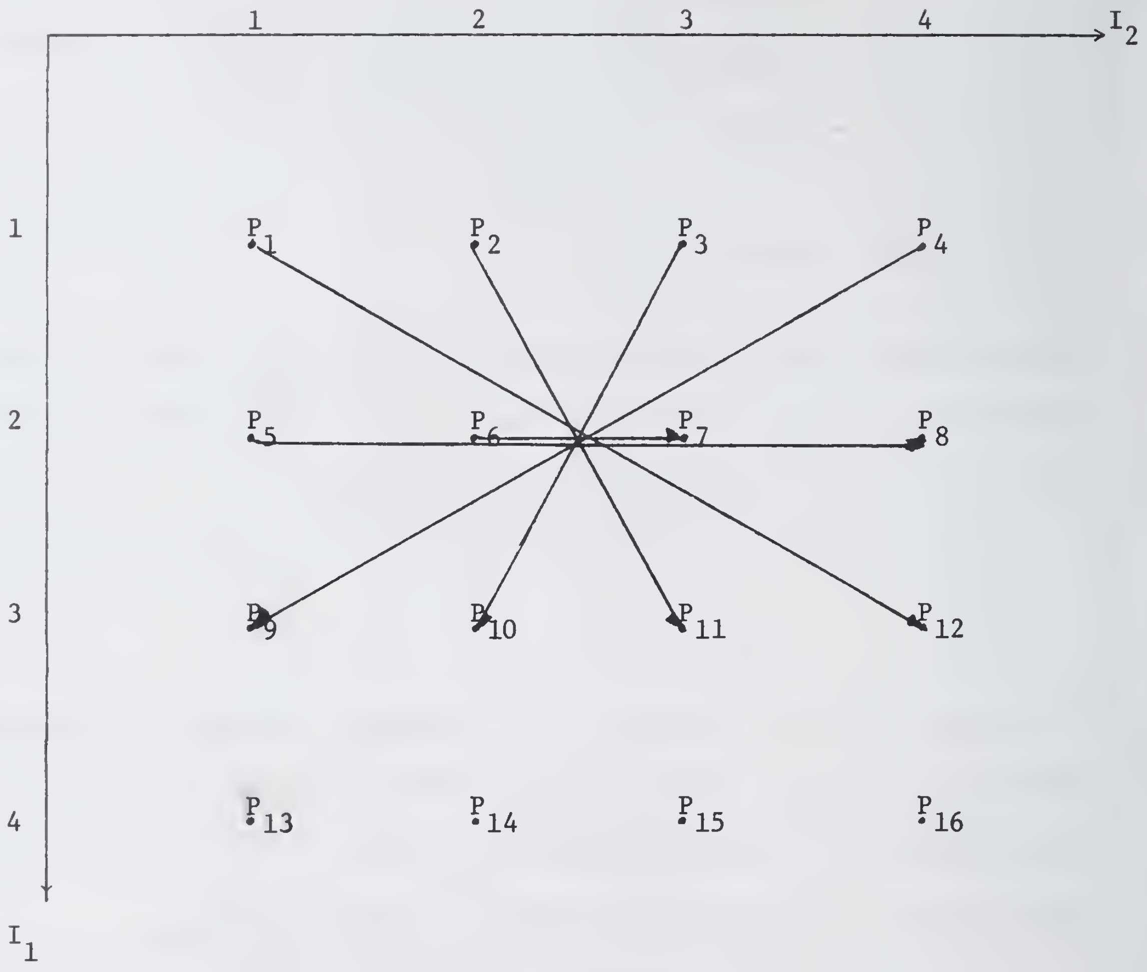


Figure 21. Dependence Vectors for Program 15

Moreover, Let

$$Y_1 = \{F_1(1), F_1(2), \ldots, F_1(N)\}$$

$$y_1 = \{f_1(1), f_1(2), \ldots, f_1(N)\}$$

$$Y_2 = \{F_2(1), F_2(2), \ldots, F_2(N)\}$$

$$y_2 = \{f_2(1), f_2(2), \ldots, f_2(N)\}$$

Then the page indexing transformation cannot be applied to this program if and only if both of the following conditions are true:

<u>C1</u>:

$$T_1 = Y_1 \cap y_1 \neq \phi = \{F_1(k_{11}), F_1(k_{12}), \ldots, F_1(k_{1m})\} =$$
$$\{f_1(k_{21}), f_1(k_{22}), \ldots, f_1(k_{2m})\}$$

and there exists k_{1p} and k_{2p}, $1 \leq p \leq m$ such that $k_{1p} < k_{2p}$. Note that $F_1(k_{1p}) = f_1(k_{2p}) \in T_1$.

<u>C2</u>:

$$T_2 = Y_2 \cap y_2 \neq \phi = \{F_2(j_{11}), F_2(j_{12}), \ldots, F_2(j_{1\ell})\} =$$
$$\{f_2(j_{21}), f_2(j_{22}), \ldots, f_2(j_{2\ell})\}$$

and there exists j_{1q} and j_{2q}, $1 \leq q \leq \ell$ such that $j_{1q} > j_{2q}$. Note that $F_2(j_{1q}) = f_2(j_{2q}) \in T_2$.

<u>Proof:</u>

The theorem states that the combined condition $C = C1 \cdot C2$ is a necessary and sufficient condition for the page indexing transformation not to be valid. This is equivalent to saying that the page indexing transformation is valid if and only if C1 is not true or C2 is not true.

If T_1 is an empty set then C1 will not be true. Note that since F_1 and f_1 are functions only of I_1 then the program will write in a particular row of A using only points from a single row. Thus if $T_1 = \emptyset$, then when the program is writing in a row of A it will read values from points on a row which was never (and will never be) written into. Thus there will be no data dependence vectors between any two points of the $I_1 x I_2$ plane. This means that the cursor can visit the points in the $I_1 x I_2$ plane in any order, and hence the page indexing transformation will be valid.

C2 will not be satisfied if T_2 is empty. By an argument similar to the one presented in the previous paragraph, if $T_2 = \emptyset$ there will be no data dependence vectors between any two points of the $I_1 x I_2$ plane. Thus the transformation will be valid.

If $T_1 \neq \emptyset$ and $T_2 \neq \emptyset$ then dependence vectors may exist. Consider Figure 22. When the cursor is at point P (k_{2p}, j_{2q}) (i.e., the program is assigning a value to $A(F_1(k_{2p}), F_2(j_{2q}))$, the value of $A(f_1(k_{2p}), f_2(j_{2q}))$ will be used on the right-hand side of S. If $f_1(k_{2p}) \varepsilon T_1$ and $f_2(j_{2q}) \varepsilon T_2$, then there must exist k_{1p} such that $F_1(k_{1p}) = f_1(k_{2p})$ and j_{1q} such that $F_2(j_{1q}) = f_2(j_{2q})$. Thus there will exist a vector from the point $P_x = (k_{1p}, j_{1q})$ to the point $P = (k_{2p}, j_{2q})$. Let θ be the angle between the vector drawn from P_x to P and the I_2 direction. As shown in Figure 22, θ can take any value between 0° and 360°. From our previous description of the manner in which the cursor will travel in $I_1 x I_2$ when page indexing is used (see Figure 20-b) we conclude the <u>transformation will be valid</u> for all page sizes if and only if

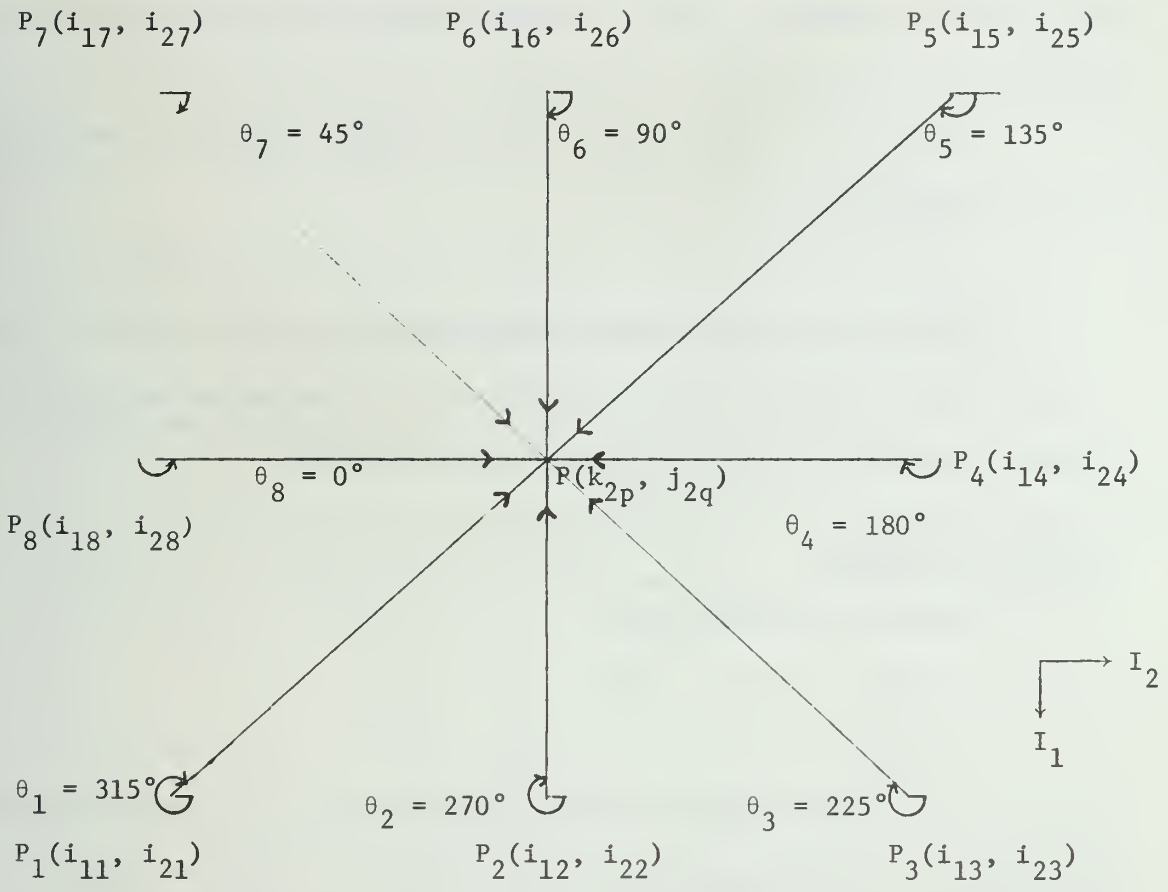


Figure 22. Dependence Vectors in the I_1xI_2 Plane

$0° \le \theta \le 90°$ or $180° \le \theta \le 360°$. In other words, the transformation will not be valid if and only if $90° < \theta < 180°$.

For $90° < \theta < 180°$, $\sin\theta > 0$ and hence $k_{2p} - k_{1p} > 0$. Moreover $\cos\theta < 0$ and hence $j_{2q} - j_{1q} < 0$.

Q.E.D.

Fince F_1 and f_1 are linear functions of I_1 and similarly F_2 and f_2 are linear functions of I_2 the following theorem can be used to test whether condition C1 or C2 of Theorem 3 is satisfied.

Theorm 3.3 [BANE78]:

Given the two functions

$f(I) = \alpha + aI$ and

$g(I) = \beta + bJ$

where α, β, a, b are integer constants ($\neq 0$), and I is an integer variable such that $1 \le I \le N$, then the two sets $\{f(1), \ldots, f(N)\}$ and $\{g(1), \ldots, g(N)\}$ intersect and there will be at least two integers i, j such that $f(i) = g(j)$ with $i < j$, if and only if the following conditions are satisfied.

(A) $\gcd(a, b) = d$ divides $\beta - \alpha$; and

(B) $\lceil \max U(i_o, j_o) \rceil \le \lfloor \min V(i_o, j_o) \rfloor$

where

(i) gcd(a, b) is the greatest common divisor of a and b.

(ii) $(i = i_o,\ j = j_o)$ is any solution to the equation

$ai - bj = \beta - \alpha$

(iii) the two sets $U \equiv U(i_o, j_o)$ and $V \equiv v(i_o, j_o)$ are defined as follows:

$(1 - i_o)*d/b$ is in U if $b > 0$, in V if $b < 0$;

$(N - i_o)*d/b$ is in U if $b < 0$, in V if $b > 0$;

$(1 - j_o)*d/a$ is in U if $a > 0$, in V if $a < 0$;

$(N - j_o)*d/a$ is in U if $a < 0$, in V if $a > 0$.

$(i_o - j_o+1)*d/(a-b)$ is in U if $a > b$, in V if $a < b$.

<u>Proof</u>: see [BANE78]

We now illustrate the use of Theorem 3.3 in testing C1 and C2 of Theorem 3.2. Consider the following program:

```
            DO   S  I1 = 1, 9
            DO   S  I2 = 1, 9
S           A(2I1-1, I2+2) = A(I1+1, 10-I2)
```

We first check if C1 is true. Thus we test whether the two functions

$f(I) = 2I - 1 = aI + \alpha$ and

$g(J) = J + 1 = bJ + \beta$

will intersect and whether there is some i and j such that $f(i) = g(j)$ and $i < j$. The $\gcd(a, b) = 1$ and $\beta - \alpha = 2$. Thus $\gcd(a,b)$ divides $\beta - \alpha$. A particular solution to the equation $2I - J = 2$ is $i_o = 10$ and $j_o = 18$. U is the set $(-9, -8.5, -7)$ and the set V is $(-1, -4.5)$. $\lceil \max U(i_o, j_o) \rceil = -7$ and $\lfloor \min V(i_o, j_o) \rfloor = -5$. Hence condition (B) is satisfied.

Thus C1 is true of this program. Now we test whether C2 also holds. Thus we test whether the two functions

$f(I) = -I + 10 = aI + \alpha$ and

$g(J) = J + 2 = bJ + \beta$

intersect at some $I = i_1$ and $J = j_1$ such that $i_1 < j_1$.

Here we have

$$\alpha - \beta = 8$$

$$\gcd(a, b) = 1$$

Hence condition (A) is satisfied. A particular solution to the equation $-I - J = -8$ is $i_o = 2$ and $j_o = 6$. Thus we have:

$$U = (-1, -3), \lceil \max U(i_o,j_o) \rceil = -1$$

$$V = (7, 5, \tfrac{3}{2}), \lfloor \min V(i_o,j_o) \rfloor = 1$$

Hence condition (B) is also satisfied and C2 holds for this program. Since both C1 and C2 are true,page indexing cannot be applied to this program.

In Theorem 3.3 we assumed that $a \neq 0$, $b \neq 0$, and $a-b \neq 0$. The conditions to be tested are simple if these assumptions did not hold. For example, if $a \neq 0$ and $b = 0$ then the two functions $f(I) = aI + \alpha$ and $g(J) = \beta$ will intersect if and only if $\frac{\beta - \alpha}{a}$ is an integer between 1 and N. In the case where $a = b \neq o$ the two functions will intersect if $\frac{\alpha - \beta}{b}$ is an integer between 1 and N. For this case the two functions $f(I) = bI + \alpha$ and $g(J) = bJ + \beta$ will intersect at the points $(I = i, J = i + \frac{\alpha - \beta}{b})$, $i = 1, 2, \ldots, N - \frac{\alpha - \beta}{b}$.

If different elements of the array A are referenced in the right-hand side of the statement S in the NP under consideration, then we use Theorems 3.2 and 3.3 to determine whether C1 and C2 hold between the subscript expression of the output variable $A(F_1(I_1), F_2(I_2))$ and the subscript expressions of every reference to a different element of A on the right-hand side of S. If the π-block has more than one statement then we do the testing between the set of multi-dimensional array output variables and all references to different elements of these arrays in the set of input variables of the π-block. Note that for the π-block:

```
DO  $S_m$  $I_1 = 1, N_1$
DO  $S_m$  $I_2 = 1, N_2$
    $S_1$
    $S_2$
    .
    .
    .
    $S_m$
```

the set of output variables is $\bigcup_{k=1}^{m} \mathrm{OUT}(S_k(I_1, I_2))$ and the set of input variables is given by:

$$\bigcup_{k=1}^{m} [\mathrm{IN}(S_k(I_1, I_2)) - \bigcup_{\ell=1}^{k-1} \mathrm{OUT}(S_\ell(i_1, i_2))]$$

If the basic NP has several π-blocks we must do the testing between the set of multi-dimensional output variables of the NP and their occurrences in its set of input variables.

3.5.3 Transforming Nonbasic π-Blocks into Basic π-Blocks

The page indexing algorithm does not achieve its goals if applied to nonbasic π-blocks. As an example consider the following program:

Program 16-a.

```
DI      DO  S2  I = 1, N
S1      B(I,1) = A(I,1)**.5
DJ      DO  S2  J = 1, N
S2      A(I+1, J) = B(I, J) + C(I, J)
```

If we apply the page indexing algorithm as described in the previous section to Program 16-a, we get the following program:

Program 16-b.

```
        DO  10  IP = 1, ⌈N/RZ⌉
        ILB = 1 + (IP-1)*RZ
        IUB = MIN(IP*RZ, N)
        DO  10  I = ILB, IUB
S1      B(I,1) = A(I,1)**.5
        DO 10 JP = 1, ⌈N/RZ⌉
        JLB = 1 + (JP-1) * RZ
        JUB = MIN (JP * RZ,N)
        DO 10 J = JLB, JUB
S2      A(I+1,J) = B(I,J) + C(I,J)
10      CONTINUE
```

We note that the index sequencing of Program 16-b is identical to that of Program 16-a. The advantages of page indexing, i.e., making the maximum number of references to a page while it is in main memory are not achieved.

Any nonbasic π-block, however, can be changed to a basic one by expanding the scope of some of its DO statements to make all assignment statements fall at the same nest depth level. Of course some of these statements must now be executed conditionally. For example the scope of the DJ statement in Program 16-a can be expanded to include S_1. The resulting basic π-block is shown below.

Program 16-c.

```
DI    DO    S2 I=1,N
DJ    DO    S2 J=1,N
S1          IF(J.EQ.1)  B(I,1) = A(I,1)**.5
S2          A(I+1,J) = B(I,J) + C(I,J)
```

The page indexing transformation will now be effective. This is shown below.

Program 16-d.

```
            DO     S2   IP = 1, ⌈N/RZ⌉
            ILB = 1 + (IP-1)*8
            IUB = MIN(IP*8,N)
            DO     S2   JP = 1, ⌈N/RZ⌉
            JLB = 1 + (JP-1)*8
            JUB = MIN(JP*8,N)
            DO     S2   I = ILB, IUB
            DO     S2   J = JLB, JUB
S1          IF(J.EQ. JLB.AND.JP.EQ.1)
                             B(I,1) = A(I,1)**.5
S2          A(I+1,J) = B(I,J) + C(I,J)
```

Note the modification in the IF statement.

We now discuss a general algorithm to transform any π-block structure into a basic structure. Let the set of DO statements in the π-block be $D\pi = \{DI_1, DI_2, \ldots, DI_{ND}\}$, $ND > 1$. The set of corresponding index variables is denoted by $\{I_1, I_2, \ldots, I_{ND}\}$. For each DO statement DI_j we denote the lower bound of its index variable set by L_j and the upper bound by U_j. Let the set of non-DO statements in the π-block be denoted by $S\pi = \{S_1, S_2, \ldots, S_m\}$, $m > 1$. For each S_i let

DB_i = {the set of DO statements that precede S_i and whose scope do not include S_i}

$= \{DI_{bi,1}, DI_{bi,2}, \ldots, DI_{bi,k_i}\}$, $0 \leq k_i < ND$.

Moreover, let

DA_i = {the set of DO statements that follow S_i}

$= \{DI_{ai,1}, DI_{ai,2}, \ldots, D_{ai,s_i}\}$, $0 \leq s_i < ND$.

Then the π-block can be transformed to the form:

DI_1

DI_2

$\vdots$

DI_{ND}

$S_1 . C_1$

$S_2 . C_2$

$\vdots$

$S_m . C_m$

where C_i is a Boolean variable which controls the execution of S_i . If C_i is true then S_i is executed,else it is not. C_i is given by:

$$C_i = \{(I_{bi,1}=U_{bi,1}).\text{AND}.(I_{bi,2}=U_{bi,2})\ldots.\text{AND}.(I_{bi,k_i}=U_{bi,k_i}).\text{AND}.$$
$$(I_{ai,1}=L_{ai,1}).\text{AND}.(I_{ai,2}=L_{ai,2})\ldots\ldots.\text{AND}.(I_{ai,s_i}=L_{ai,s_i})\}$$

To illustrate the application of this algorithm consider the Gaussian elimination program shown below:

Program 17-a.

```
DI$_1$     DO   S$_2$  I$_1$ = 1,N-1
DI$_2$     DO   S$_1$  I$_2$ = (I$_1$+1),N
S$_1$      A(I$_2$,I$_1$)=A(I$_2$,I$_1$)/A(I$_1$,I$_1$)
DI$_3$     DO   S$_2$  I$_3$ = (I$_1$+1),N
DI$_4$     DO   S$_2$  I$_4$ = (I$_1$+1),N
S$_2$      A(I$_4$,I$_3$)=A(I$_4$,I$_3$)-A(I$_4$,I$_1$)*A(I$_1$,I$_3$)
```

Here we have:

$D\pi = \{DI_1, DI_2, DI_3, DI_4\}$

$S\pi = \{S_1, S_2\}$

$DB_1 = \phi$

$DA_1 = \{DI_3, DI_4\}$

$DB_2 = \{DI_2\}$

$DA_2 = \phi$

$C_1 = I_3.EQ.(I_1+1).AND.I_4.EQ.(I_1+1)$

$C_2 = I_2.EQ.N$

Thus the corresponding basic π-block is as follows:

Program 17-b.

```
DI$_1$     DO   S$_2$   I$_1$=1,N-1
DI$_2$     DO   S$_2$   I$_2$=(I$_1$+1),N
DI$_3$     DO   S$_2$   I$_3$=(I$_1$+1),N
DI$_4$     DO   S$_2$   I$_4$=(I$_1$+1),N
S$_1$      IF   (I$_3$.EQ.(I$_1$+1).AND.I$_4$.EQ.(I$_1$+1))
                          A(I$_2$,I$_1$) = A(I$_2$,I$_1$)/A(I$_1$,I$_1$)
S$_2$      IF   (I$_2$.EQ.N)
                     A(I$_4$,I$_3$) = A(I$_4$,I$_3$) - A(I$_4$,I$_1$)*A(I$_1$,I$_3$)
```

We note that this algorithm will introduce a large amount of control instructions when the π-block is executed. This excessive control can be reduced by fusing some of the loops in the π-block, whenever possible, before expanding their scopes. Note that at this point in the transformation process we know the data dependences in the π-block and thus checking for the validity of loop fusion is a trivial additional expense.

The combined loop expansion-fusion transformation can be applied to Program 17-a in the following steps:

(Expand DI_3)

Program 17-c.

```
DI1     DO   S2      I1 = 1,(N-1)
DI3     DO   S2      I3 = (I1+1),N
        IF (I3.EQ.I1+1)
DI2     ⎧DO   S1   I2 = (I1+1),N          ⎫
S1      ⎩A(I2,I1) = A(I2,I1)/A(I1,I1)     ⎭
DI4     DO   S2      I4 = (I1+1),N
S2      A(I4,I3) = A(I4,I3) - A(I4,I1)*A(I1,I3)
        (Fuse DI2 and DI4)
```

Program 17-d.

```
DI1     DO   S2      I1 = 1,(N-1)
DI3     DO   S2      I3 = (I1+1),N
DI2     DO   S2      I2 = (I1+1),N
```

```
S1      IF (I3.EQ.I1+1) A(I2,I1) = A(I2,I1)/A(I1,I1)
S2      A(I2,I3) = A(I2,I3) - A(I2,I1)*A(I1,I3)
```

Thus in general, the nonbasic to basic π-block transformation consists of a series of loop expansion and fusion steps. One starts by trying to fuse loops in the given π-block. This is followed by expanding the scope of the farthest reaching DO statement (if we associate a CONTINUE statement with each DO statement and number these CONTINUE statements sequentially, then the farthest reaching loop is the one associated with the CONTINUE statement with the largest label). This process of fusion followed by expansion is continued until a basic π structure is reached. Note that to expand a loop we use the algorithm presented previously in this section.

For Program 17-d the page indexing transformation can now be applied as shown below. (This is a legal Fortran version. Also note that we have substituted K for I_1, J for I_3, and I for I_2).

```
Program 17-e
RZ = Z ** .5
NP =⌈N/RZ⌉
DO  S2  KP = 1, NP
KLB = 1 + (KP - 1) * RZ
DO  S2  JP = KP, NP
JLB = 1 + (JP - 1) * RZ
JUB = JP * RZ
DO  S2  IP = KP, NP
ILB = 1 + (IP - 1) * RZ
```

```
         IUB   = IP * RZ
         IF (IP.EQ.KP) KUB = KP * RZ - 1
         IF (IP.NE.KP) KUB = KP * RZ
         DO  S2  K = KLB, KUB
         IF (IP.EQ.KP) ILB = K + 1
         IF (JP.EQ.KP) JLB = K + 1
         DO  S2  J = JLB, JUB
         DO  S2  I = ILB, IUB
         IF (J.EQ.JLB.AND.JP.EQ.KP) A(I,K) = A(I,K)/A(K,K)
S2       IF (J.LE.JUB) A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

4. EXPERIMENTAL RESULTS

The aim of this chapter is to provide some preliminary experimental evidence of the usefulness of the transformations presented in Chapter Three. We will also discuss some experiments which we performed to investigate the concept of bounded locality intervals [BATS76a] and the correlation between a program's syntactic structure and its BLI's

We have chosen 17 Fortran IV programs to experiment with. There were two reasons to select programs written in Fortran and not in other languages. First, there are a large number of all kinds of Fortran programs available for experimentation. Second, the current version of the PARAFRASE compiler accepts only Fortran programs. We think of the transformations presented in Chapter Three as modifications and extensions to some of the transformations already existing in the PARAFRASE compiler in addition to some new ones which are specifically aimed at the enhancement of the performance of virtual memory systems.

Eleven of our programs were chosen from a collection of programs which we got from different national laboratories. In the other six programs we coded some standard matrix algorithms. In selecting the eleven programs we followed two guidelines. First, we wanted a set of programs which was fairly representative of various numerical Fortran programs. We wanted the complexity of the calculations performed in the programs to vary from simple or merely data movement operations to complex computations. Second, we eliminated any programs which have relatively small memory requirements. We required that each of the chosen programs has a virtual address space of more than twenty pages.

We have chosen the page size to be 256 bytes (64 words). For our purposes, the choice of the page size is not critical. We are trying to demonstrate that programs which reference multi-page arrays, irrespective of the size of one page, can be transformed to behave better in a paged virtual memory environment. At the end of this chapter we will discuss the effect of varying the page size on our results. We will show that the effectiveness of our transformations is rather independent of the page size. For our purposes, what matters is not the absolute value of the size of pages and the sizes of arrays but their relative sizes. Since we are mostly interested in programs which have large virtual space (these are the programs which usually can have disasterous behavior in virtual memory machines) a page size of 256 bytes seemed to be suitable to ensure that our collection of programs have large space requirements. As mentioned earlier we will return to this subject in much more detail at the end of this chapter.

Table 7 shows a brief description of the programs used in our experiments. The total number of source cards (excluding comments) is 1598. The total number of DO statements is 200. We generate the trace of a program using the arrangement shown in Figure 23. The input Fortran program is passed through the scanner of the PARAFRASE compiler and the IBM Fortran IV G1 level 2.0 compiler. The output of the Fortran compiler is a listing showing every statement of the source program and the portion of the object code associated with it. We examine this output and make a list of the statement numbers of those statements which must be executed by the trace generator. These include any statements which

Table 7. The Programs Used in the Experiments.

Program	Source	# of State-ments	# of DO State-ments	# of Arrays	# of Array Refer-ences	# Pages Refer-enced	Comments
ADVECT	UIARL*	69	6	19	63 744	226	Cloud Physics.
BASE	UIARL	144	12	29	21 747	300	Cloud Physics.
BIGEN	AFWL*	36	3	7	41 997	98	Initialization.
CD	Coded	20	4	1	40 424	27	Cholesky Decomposition (48 x 48 System).
DISPERSE	NSF*	244	26	52	23 659	734	Chemical Analysis.
FIELD	AFWL	66	9	20	11 152	52	Electromagnetic Fields.
FLR	Coded	13	2	3	4 608	23	Full Linear Reucrrence (48 x 48 System).
FOURTR	NRL*	50	10	7	86 012	128	Fast Fourier Transform (1024 Points).
GE	Coded	14	4	1	146 264	36	Gaussian Elimination (48 x 48 System).
INIT	AFWL	55	14	25	12 154	245	Initialization.
LUD	Coded	26	5	1	77 224	36	LU Decomposition (48 x 48 System).
MAIN	UIARL	245	33	38	81 792	198	Cloud Physics.
MAMOCO	NRL	35	6	8	236 027	875	Matrix Mode Coupling.
MATMUL	Coded	11	3	3	257 600	75	Matrix Multiplication (40 x 40 Matrices).
MATTRP	Coded	11	2	1	3 276	25	Matrix Transpose (in Place - 40 x 40).
PAPUAL	NRL	47	5	6	58 688	1 418	Random Particle Velocity Genator.
TWOWAY	UIARL	512	56	57	151 189	282	Cloud Physics.

*UIARL - University of Illinois Atmospheric Research Laboratory
*AFWL - Air Force Weapons Laboratory
*NSF - National Science Foundation
*NRL - Naval Research Laboratory

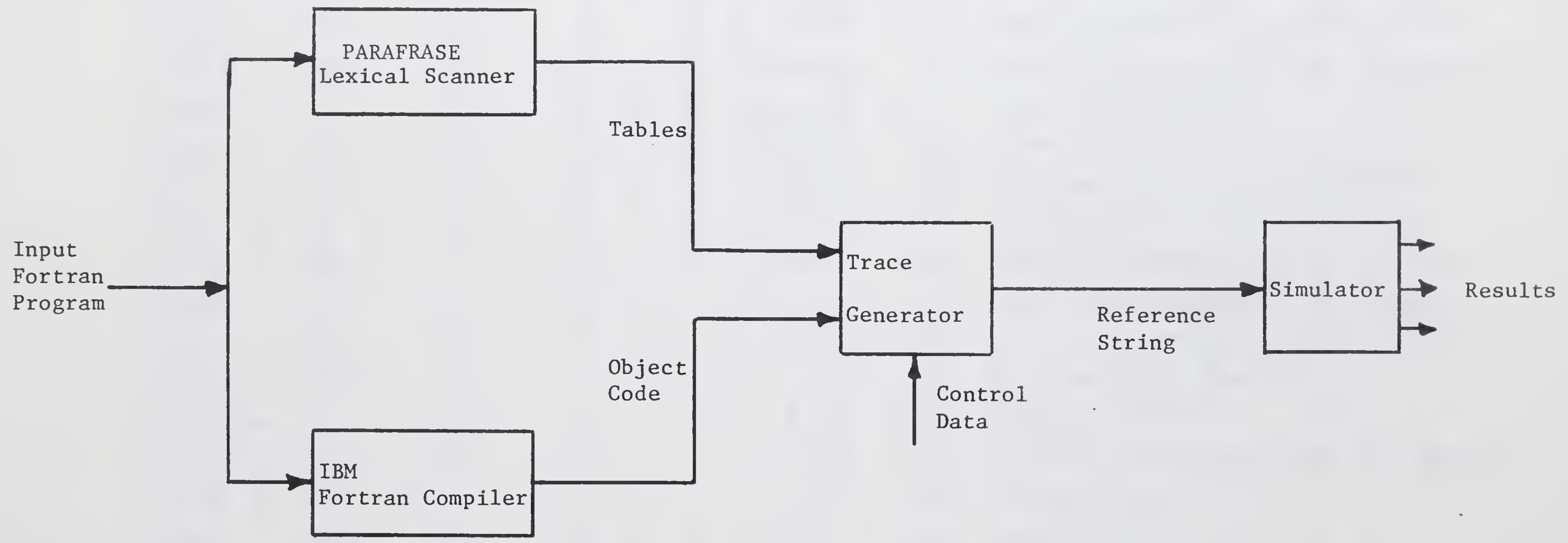


Figure 23. The Scheme Used to Generate Reference Strings and Simulation Results.

calculate index variables, loop bounds, or conditions of logical IF statements. The trace generator receives the output of the Fortran compiler, the program description tables from the PARAFRASE scanner, and a control input which includes the list of statements to be executed, specification of the storage scheme of multi-dimensional arrays (storage by rows, columns, or square blocks), the page size in words, necessary values for some variables used in the input program, and branching probabilities to be used in those IF statements for which the test condition cannot be evaluated by the current trace generator. Thus the trace generator will simulate a partial execution of the input program which is sufficient to get an accurate trace of array references. The branching probabilities and the values to be given to variables are chosen with the help of the documentation of the input program or by personal communication with the people who supplied the program.

In two occasions we had to eliminate a loop in a program. We eliminated the following loop from the Fast Fourier Transform program, FOURTR:

```
DO   S6    I = 1, N2N
S1   IIN1 = 1 + REVERS (I)
S2   IIN2 = 1 + REVERS (I+1)
S3   CR(I)= INR(IIN1) + INR(IIN2)
S4   CI(I)= INI(IIN1) + INI(IIN2)
S5   CR(I+1) = INR(IIN1) - INR(IIN2)
S6   CI(I+1) = INI(IIN1) - INI(IIN2)
```

This had to be done because the current trace generator cannot evaluate statements S_1 and S_2 which is necessary to calculate some subscripts in statements S_3, S_4, S_5, and S_6. The current trace generator does not evaluate expressions if they contain array elements.

In the other occasion we eliminated a loop from the TWOWAY program. This loop contained 211 statements with several inner loops at different nest depth levels. Analyzing this program with this loop included exceeded the capabilities of the current PARAFRASE compiler.

Our original plans for the experiments were to apply one transformation at a time to each of our programs in order to measure the contribution of each transformation to the total achieved improvement. We decided to abandon these plans for the time being due to the enormous amount of results which would be generated. Thus we applied all the transformations possible to a given program in order to achieve the best possible improvement. We used a mixture of automatic and manual means for applying the transformations. The data dependence relations were analyzed automatically. Part of the transformations were already implemented in the PARAFRASE compiler. The clustering transformation has been added to PARAFRASE and work is continuing to add the rest of the transformations. To obtain our current results, whenever we had to, we applied the transformations manually. We would like to emphasize that we look at the experimental results reported here as preliminary results. We decided that initially it is important to get a feeling for the amount of improvement which can be achieved in the behavior of real programs by transforming a few programs, using automatic and manual means, and examining the results rather than waiting to fully automate the transformations before generating any results. We feel that our preliminary results serve as the green light which signals that the investment of effort in automating all our techniques is a safe investment.

In Table 8 we compare some of the characteristics of the original and transformed programs. This table is meant to give a feeling for the worst possible cost of transforming a program. We will explain as our discussion progresses why this is the worst cost of the transformations. For 6 of the 17 programs the number of pages referenced in the transformed program exceeds the number in the original program. This is due to the scalar expansion transformation. We note that the maximum increase is 5 pages. We also notice an increase in the number of array references for those programs where scalar expansion was used. This is not a real increase in the number of memory references to data words in the transformed program. These extra memory references reported for the transformed program are also made in the original program, but to scalar variables. For the original programs these references were just <u>not counted</u> because we only count references to array elements. The increase shown in the number of source statements in the transformed programs is not really accurate. It is an over estimate. The reason for this is that our current trace generator is not very smart and in many cases we had to insert redundent statements to make the tracer do what it is supposed to do. For example the current tracer cannot evaluate ILB in the following statement:

```
      IF (KP.EQ.1)   ILB = K+1
```

To achieve this assignment to ILB we do the following

```
      IF (KP.NE.1)   GO TO 1
      ILB = K + 1
    1 ......
```

Moreover, our tracer does not evaluate functions. Thus to make the assignemnt:

Table 8. Some Characteristics of the Original and Transformed Programs.

Program	# Pages Referenced		# Array References		# Source Statements		# Instructions Executed	
	Original	Transformed	Original	Transformed	Original	Transformed	Original	Transformed
ADVECT	226	231	63 744	123 297	69	139	584 472	722 009
BASE	300	301	21 747	24 143	144	246	91 529	214 576
BIGEN	385	385	41 997	41 997	36	41	129 563	210 054
CD	21	23	40 424	79 528	20	43	234 211	2 202 748
DISPERSE	734	734	23 659	23 659	244	319	219 182	310 171
FIELD	52	52	11 152	11 152	66	112	31 435	46 398
FLR	23	23	4 608	4 608	13	31	14 884	18 160
FOURTR	128	128	86 012	101 376	50	56	413 335	556 537
GE	36	36	146 262	146 264	14	34	494 314	1 619 039
INIT	245	245	12 154	12 154	55	109	92 208	143 009
LUD	36	38	77 224	153 272	26	47	507 543	2 247 035
MAIN	198	200	81 792	102 497	245	253	444 591	685 406
MAMOCO	875	875	236 027	236 027	35	48	2 458 302	2 600 695
MATMUL	75	75	257 600	257 600	11	28	790 971	1 071 584
MATTRP	25	25	3 276	3 276	11	23	12 178	39 889
PAPUAL	1418	1418	58 688	58 688	47	81	1 311 592	1 601 939
TWOWAY	282	285	119 149	126 797	512	1095	484 145	733 536

```
         IUB = MIN (N,IP*Z)
```

we do the following

```
         IUB = IP*Z
         IF(IUB.LE.N) GO TO 2
         IUB = N
     2   ....
```

These and other inefficiencies in our tracer lead also to an over estimation of the increase in the number of instructions executed in the transformed programs. The more pronounced increase in the number of executed instructions for programs CD, LUD, and GE is mainly due to the nonbasic to basic π-block transformation. Our current implementation of this transformation introduces an appreciable amount of control instructions. Further effort needs to be made to improve the implementation of this transformation. In Chapter 5 we make some suggestions concerning this point.

Our experiments fall in three categories. In the first we implemented the algorithms described in [BATS76a] to find the BLI's of our programs and their transformed versions. The purpose of these experiments is to investigate the validity of the BLI concept in defining the localities of a program. Moreover, we wanted to compare the characteristics of the localities found in a program to those found in its transformed version. We also wanted to compare our findings to the experimental results reported in [BATS76a]. We will discuss all these issues in Section 4.1.

In the second category of experiments we simulated the local LRU memory management algorithm and generated the page-faults vs.memory allotment and the space-time cost vs. memory allotment curves of every program and its transformed version. The purpose is to compare the cost of executing original and transformed programs under LRU. We have chosen the LRU algorithm because it is known to be the best among the heuristic replacement algorithms and because most of the existing virtual memory machines use some sort of an LRU algorithm for memory management [ScHE73],[JONE72]. The results of these simulations are discussed in Section 4.2.

The third category of experiments are designed to investigate the important question of finding whether there are any merits for using variable memory allotment policies as compared to using fixed memory allotment policies for the memory management of transformed programs. We have chosen to use the working set management policy as a representative of variable memory policies [DENN68]. We compared the space-time-cost for the transformed programs under the LRU and working set policies. For several programs we encountered the real memory-fault rate and parameter-real memory anomalies as described in [FRAN78]. This point and the LRU-working set comparison will be discussed in Section 4.3.

In Section 4.4 we summarize the implication of our findings and investigate the sensitivity of our results to the page size.

4.1 Measuring the Characteristics of Program Localities

To measure the characteristics of program localities one has first to identify these localities. This can easily be done for the transformed versions of our collection of programs because they follow the ELM. In a transformed program, whenever a π-block is being executed, the reference string will stay within one locality interval. The MTBR to every page of this locality is small, $O(R_\ell)$, where R_ℓ is the number of array references made per iteration of the innermost loop of the π-block. The density of references to a page is high. Hence for transformed programs one can identify localities, count the number of pages referenced in each locality, and its duration.

Loops in untransformed programs do not in general follow the ELM and hence it is not easy to identify localities and measure their characteristics.

Thus one can measure the characteristics of localities in transformed programs but cannot compare these measurements in an accurate way to measurements made on the original programs. The localities of original programs are simply not well defined! The locality of reference of untransformed programs is a vague, loose, and unquantifiable concept.

The work of Batson and Madison [BATS76a],[BATS76b], is the only effort previously made to identify localities in reference strings of programs. In Chapter 2 we have shown that there are several problems with the concept of BLI's as developed in [BATS76a]. We confirmed the existance of these problems by implementing Batson's algorithms and finding the BLI's of our programs. We then correlated the BLI's structure of a program to its syntactic structure. We made assumptions which are identical to those made by Batson. He assumed that there is a one-to-one correspondence between array names and segment identifiers. In other words, he assumed a segmented virtual memory system in which the segment size can vary. Each array, irrespective of its size, is stored in a single segment.

After using the BLI's generated for our programs to investigate the correctness of the BLI concept and find its problems, we decided to use the resulting data for other purposes. If meaningless and misleading BLI's are discarded one can identify those BLI's that correspond to loops. Thus by carefully examining the BLI's of a transformed program one can find the duration of execution of each π-block and the number of referenced arrays. This gives the size and lifetime of true localities in transformed programs. For the untransformed programs we identified the BLI's corresponding to outermost loops and recorded their duration and number of referenced arrays. Our findings will be discussed in Section 4.1.1.

We used the same techniques discussed in the previous paragraph to collect data about the size and duration of localities for paged virtual memory systems. In this case an array, depending on its size, will span several 256 byte pages. In a transformed program, when a π-block is executed, one locality set of pages will be referenced after another. We collected data about the size and duration of localities of a program by carefully examining its BLI's when generated under a paged system assumption. For the untransformed programs we collected data about the number of pages referenced in BLI's that correspond to outermost loops. Our findings are discussed in Section 4.1.2.

4.1.1 <u>Localities in Segmented Systems</u>

Because of the kind of segmented system we have assumed in this section, we do not include any data from programs CD, FLR, GE, LUD, MATMUL, and MATTRP. Including data from these programs would have biased our findings towards localities of small sizes. In each of the programs MATTRP, LUD, CD, and GE only one segment is referenced. In FLR two segments are referenced and in MATMUL three segments are referenced. In selecting programs for his experiments Batson rejected any programs which reference less than six arrays. In Table 9 we compare some of the characteristics of his programs and our programs (excluding the six previously mentioned). We note that our programs have, on the average, fewer arrays. Hence the locality of our untransformed programs is slightly better than Batson's programs. Thus the improvement results which will be reported are on the conservative side. The results would have been even better if we had Batson's collection or programs with more arrays. This fact is emphasized by Figure 25 which will be discussed shortly.

Table 9. Comparing Some Characteristics of Our Programs and those Used by Batson and Madison.

	Our Programs	Batson and Madison Programs
Number of Arrays Referenced in a Program		
Minimum	6	6
Average	24.3	26.1
Maximum	57	127
Size of the Reference Strings		
Minimum	11 152	5 459
Average	71 651	42 857
Maximum	236 027	102 227

Before discussing our results we make one more remark. Our programs were transformed with the assumption that they will run in a paged system with a page size of 256 bytes. However, it is not difficult to deduce the characteristics of the localities if the programs were transformed to run on a variable segment size system. The only thing we have to do is to eliminate the effect of the page indexing transformation on the generated data. For example consider the following program:

Program 18-a.

```
          DO  10  I = 1,16
          DO  10  J = 1,16
          A(I,J) = B(I,J) + C(I,J)
     10   D(I,J) = B(I,J)/2
```

For a variable segment size virtual memory system this program will be transformed as follows:

Program 18-b.

```
          DO   101   I = 1,16
          DO   101   J = 1,16
    101   A(I,J) = B(I,J) + C(I,J)
          DO   102   I = 1,16
          DO   102   J = 1,16
    102   D(I,J) = B(I,J)/2
```

The resulting BLI structure is shown in Figure 24-a. We have two localities. The first includes segments A, B, and C and lasts for 768 memory references. The second locality includes segments B and D and lasts for 512 memory references. If the loop of program 18-a was in one

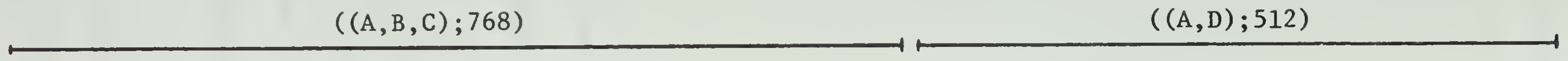


Figure 24-a. BLI's for Program 18-b.

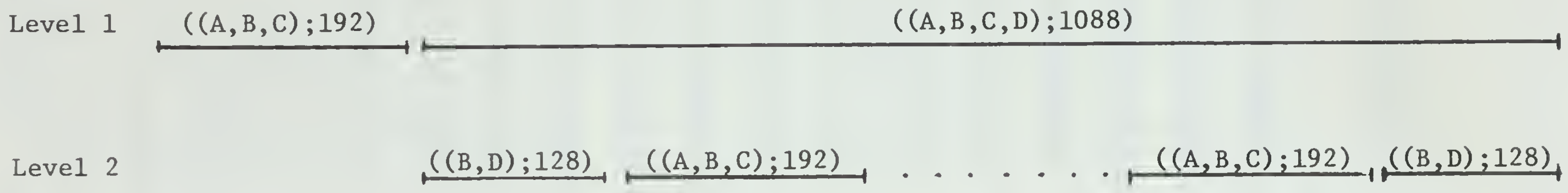


Figure 24-b. BLI's for Program 18-c.

of our collection of programs, it would have been vertically distributed as shown below:

Program 18-c.

```
          DO   10   IP = 1,2
          ILB = 1 + (IP-1)*8
          IUB = IP*8
          DO   10   JP = 1,2
          JLB = 1 + (JP-1)*8
          JUB = JP*8
          DO   101   I = ILB,IUB
          DO   101   J = JLB,JUB
  101     A(I,J) = B(I,J) + C(I,J)
          DO   102   I = ILB,IUB
          DO   102   J = JLB,JUB
  102     D(I,J) = B(I,J)/2
   10     CONTINUE
```

The BLI's of this program are shown in Figure 24-b. It can easily be seen that the localities for the segmented case in Figure 24-a can be found from those in Figure 24-b by lumping into one locality all the BLI's which have the members A,B,C. In this way we get the locality in figure 24-a with the members A,B, and C and with the duration 192*4 = 768. Similarly we get a locality of duration 128*4 = 512 and with the members B and D by simply lumping in one locality all the BLI's of Figure 24-b in which these arrays are referenced.

Figure 25 shows the characteristics of localities for the transformed programs. In the 11 programs a total of 756 121 references

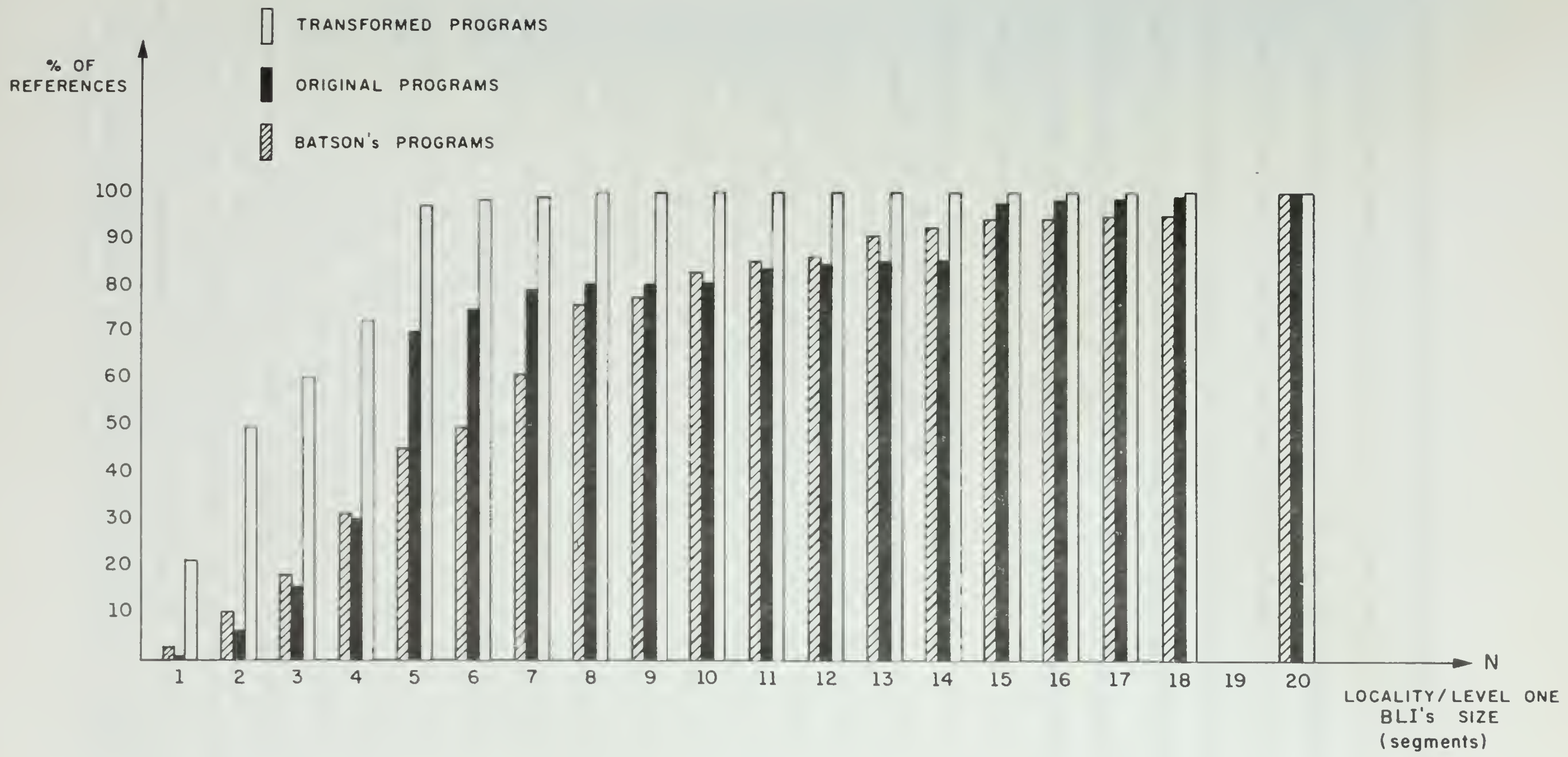


Figure 25. Percentage of Array References Made in Localities of Size $\leq$ N Segments

were made. 753 859 references were made when the programs were executing within localities. These make 99.7% of the total number of references. Part of the remaining .3% of the references were made outside loops. The other part of the .3% can be attributed to the fact that the BLI method is not exact in finding the duration of a locality. As shown in the figure more than 48% of the references were made while the transformed programs were executing within localities of size 2 or less. More than 97% of the references were made within localities of size 5 or less.

In Figure 25 we also show data for our untransformed programs and Batson's programs [BATS76a]. For Batsons' programs the figure shows the distribution of array references on level one BLI's of different sizes. For our untransformed programs the data represents the distribution of array references on BLI's which correspond to outermost loops. If we accept the argument that the data of our untransformed programs and Batson's data do not represent very different things, then one can deduce from the figure that our programs are more local than Batson's. While 45% of references are issued in level one BLI's of size less than or equal to 5 segments in Batson's programs, almost 70% of the references in our programs are made in loops with 5 or less arrays. Thus, as was mentioned earlier, our reported improvement results are on the conservative side because untransformed programs can be less local.

One can get an intuitive idea about the improvement achieved by our transformations by comparing the data of the original and transformed programs in Figure 25. Because of the assumptions made when the data was generated (one segment per array) the improvements which we see here underestimate drastically the power of the transformations. The power of the transformations will be more fairly seen when paged virtual memory systems are discussed.

As was mentioned previously, in the transformed programs more than 97% of the references are made in localities of size 5 or less. For the untransformed programs only 85% of the references are made in outermost loops with 14 or less arrays. While almost 50% of the references in the transformed programs are made in localities of size 2 or less, only 30% of the references in the original programs are made in loops with 4 or less arrays.

4.1.2 Localities in Paged Systems

The general intuitive impression one gets from examining Figure 25 is that the locality of untransformed programs is not really that bad under the assumptions of the previous section. Almost 80% of the references are made in loops with 6 or less segments (arrays). More than 98% of the references are made in loops with 15 or less segments. Since the number of segments in our programs varied between 6 and 57 with an average of 25.3, then their locality is good. One can arrive at similar conclusions from examining the data representing Batson's programs.

Virtual memory systems, however, face their serious problems when they execute programs for which the assumptions of the previous section do not hold. Batson's programs were selected from the daily work load of the University of Virginia computing center. They were executed on the B5500 computer which supports a variable segment size virtual memory system. The segment size can take values between 1 and 1023 words. Since, in his programs, there was a one-to-one correspondence between array names and segment identifiers, none of the programs had an array larger than 1023 words. Although in our programs there are many arrays which are larger than 1023 words, we still assumed that each array will occupy one segment when we generated the data of the previous section. We were interested in investigating the BLI concept and in finding a lower bound in some sense on the improvements achieved by our transformation techniques.

When multi-segment or multi-page arrays are referenced in programs, their degree of locality becomes drastically low. This is because, in general, there is no one-to-one correspondence between the number of array names referenced per iteration of a loop and the number of pages referenced. In [ELSH74] it was shown that in a paged system the locality of a matrix multiplication program which makes references only to 3 array names can be improved drastically by using some rules in accessing the elements of these multi-page arrays. Batson in [BATS76b] points out that the implications of his measurements of program localities do not apply to paged systems. We quote, "Thus it seems clear that major phases, with relatively small activity sets, span the major part of the execution epochs of programs. This phenomenon, otherwise known as locality of reference, is the raison d'être for the successful operation of symbolically-segmented virtual memory systems. <u>Its implications for paged virtual memory systems are less promising, since there is no correspondence in general between pages and symbolic segments</u>."

As we have mentioned in Chapters 2 and 3, our transformations serve two purposes. First, they make all loops behave like elementary loops for which the number of pages referenced is highly correlated to the number of array names. Thus for transformed programs there will be a one-to-one correspondence between array names and pages referenced. Second, the transformations will reduce the cost of executing programs in a paged system, namely the space-time cost, the number of page faults, and the amount of memory allotment required.

Figure 26 supports our argument. We have generated the BLI's of our 17 original programs and their transformed versions. Here we assume a paged system with a page size of 64 words (256 bytes). For the transformed programs the data in the figure represents the percentage of array references made while the programs executed with locality sizes of a particular number of pages or less. We got this data by careful correlation of the generated BLI's to the source programs. For the untransformed programs the data represents the percentage of references made while the programs executed in BLI's of sizes equal to or less than a particular number of pages. The BLI's correspond to outermost loops.

In the figure we see that for the transformed programs more than 71% of the 1 483 921 array reference were made in localities of size 3 pages or less. 83% were made in localities of size 5 pages or less and more than 97% of the references were made in localities of size 8 pages or less. If we compare Figures 25 and 26 we find that the locality of the transformed programs is comparable for both paged and segmented systems. The only noticable difference is that the percentage of references made in localities of 3 pages is higher than the percentage made in localities of 3 segments. This is because the results shown in Figure 26 include data from the six programs which we excluded from our experiments in the previous section. The transformed versions of five of these programs (CD,FLR,GE,MATNUL, and MATTRR) issue their references in localities

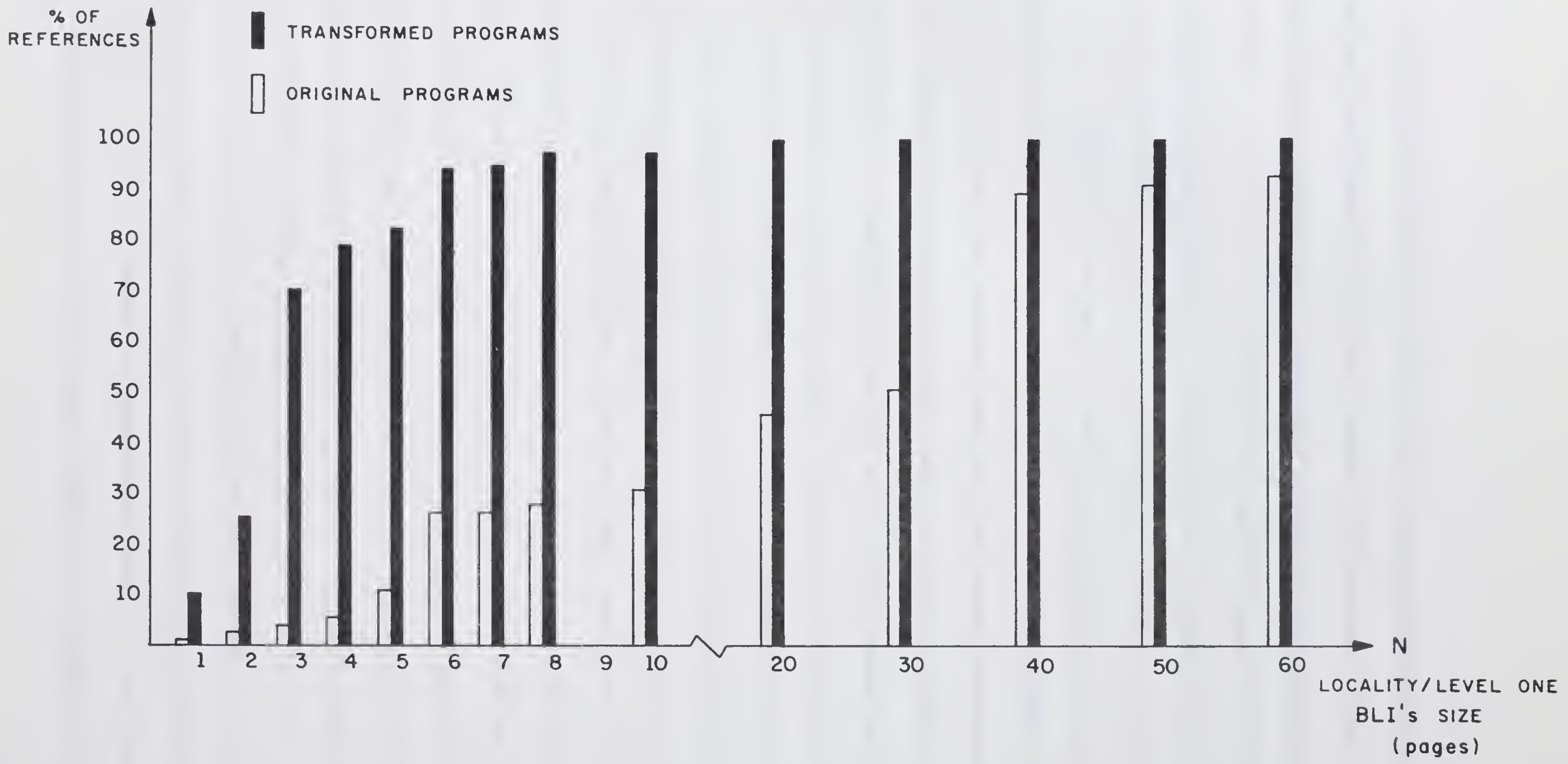


Figure 26. Percentage of Array References Made in Localities of Size $\leq$ N Pages

of 3 pages. Program LUD issues most of its references in localities of size 6 or less.

For the untransformed programs, the results shown in Figures 25 and 26 are very different. While more than 80% of the references are made in loops with 10 or less segments (array names), only 30% of the references are made when programs execute loops that correspond to BLI's of size 10 or less pages. In Figure 25 more than 98% of the references are made in loops corresponding to BLI's of size 15 segments or less. In Figure 26 only 92% of the references are made in loops that correspond to BLI's of size 60 pages or less.

Thus it is clear that the locality of untransformed programs is not as good in paged systems as it is under the assumptions of the variable segment size systems. Moreover, Figure 26 proves that our transformations have succeeded in establishing a one-to-one correspondence between the number of array names referenced in a locality and the number of pages referenced. Figure 26 also shows the appreciable reduction in the size of program localities achieved by the transformations.

4.2 Measuring the Performance Improvement of Paged Virtual Memory Systems - the Fixed Memory Allotment Case

In this section we discuss our experimental results which compare the performance of a virtual memory computer when it executes untransformed programs to its performance when it executes transformed programs. We have measured the number of page faults generated as a function of memory allotment for all our programs and their transformed versions. The replacement algorithm used was the LRU algorithm. All these page faults curves are included in the Appendix. These curves, as discussed in Chapter 2, are relevant to measuring the performance of monoprogrammed systems.

To make comparisons for a multiprogrammed machine, we have measured the space-time cost for the original and transformed programs as a function of memory allotment. These curves are also included in the Appendix.

In Section 4.2.1 we discuss the page fault curves and in Section 4.2.2 we discuss the space-time cost curves.

4.2.1 The Page Faults vs. Memory Allotment Results

If the total memory requirement of a program is less than or equal to the physical primary memory available on a monoprogrammed machine, then there is really no advantage to using a virtual memory operating system over a nonvirtual memory system in handling the memory management problem of the machine. The program simply gets all the memory it needs under both systems. The virtual memory system, however, is superior to the non-virtual memory system when programs are to be executed with memory needs which exceed the available primary memory size of the machine. In the non-virtual memory system the programmer must manually take care of the overlay problem, i.e., moving parts of the code and data of his program between primary and secondary memories during the execution of his program. In a virtual memory machine, however, this is done automatically by the OS.

A paged virtual memory system moves parts of the code and data of a program (called pages) between the different levels of a memory hierarchy when page faults occur. Thus the amount of information transferred between secondary and primary memory is proportional to the number of page faults.

Here we are mainly interested in comparing the performance of two monoprogrammed virtual memory machines. The first runs our original untransformed programs and the second runs our transformed programs. Both use the same replacement algorithm, LRU. How much better the second machine does can be used as a measure of the power of our transformation techniques. The comparison can be done in two ways. First, the two machines can be given the same amount of primary memory and then the number of page faults can be compared. In the second approach one can ask for the same performance, goodness, or efficiency from both machines and then compare the amount of primary memory which must be installed on each machine to achieve this given level of performance. One needs, however, to define a figure of merit to measure the level of performance. It seems that the ratio of the number of distinct pages to the number of page faults would be the appropriate figure of merit. Thus if the total number of distinct pages referenced during the execution of a program is DP, then the level of performance of a monoprogrammed system with a given amount of primary memory, m, can be defined as follows:

$$\text{Level of performance, } LP(m) = DP/f(m) \leq 1$$

where f(m) is the number of page faults with m page frames of primary memory.

Comparing the page faults as a function of memory allotment for our two machines, the first executing the untransformed programs and the second executing the transformed programs, can be done by examining the page fault curves of the untransformed and transformed programs in the Appendix. A full appreciation of the amount of improvement can only be achieved by examining and commenting on the curves of each program individually. We will not go into such a discussion, however, because it would be very lengthy and we will leave it to the reader to decide how much time he wants to spend staring at the curves and drawing conclusions. We will instead present in Table 10 an overview of the improvement achieved for the memory range between 4 and 8 pages. We have chosen this memory range because as discussed in Section 4.1, programs with good locality will spend most of their execution time in localities of sizes between say 3 or 4 and 8 pages. Testing of the effectiveness of the transformations is fairest when it is done in such a memory range. Note that in general with any given memory allotment a transformed program should never generate more page faults than the original program. There are two exceptions to this general rule. In the first case, due to scalar expansion, the transformed program will generate more page faults at very low memory allotment (when it is thrashing) and also with large memory allotment (because the number of distinct pages referenced in the transformed program will be larger). For this case, the additional page faults of the transformed program are not real. This increase would not exist if we counted references to scalar variables in the original program. In the second case, due to loop fusion in the nonbasic to basic π-block transformation, the transformed program may generate more page faults at low memory allotments. Let us remember that when transforming nonbasic π-blocks, loop fusion is mainly used to reduce the number of extra control instructions generated. This is done whether the fused

Table 10. The Ratio of the Number of Page Faults of the Original to the Transformed Programs for $4 \leq m \leq 8$.

Program \ Ratio	$f(4)/f_t(4)$	$f(5)/f_t(5)$	$f(6)/f_t(6)$	$f(7)/f_t(7)$	$f(8)/f_t(8)$
ADVECT	3.58	19.26	36.78	33.71	34.40
BASE	29.56	54.01	55.11	55.73	55.26
BIGEN	36.93	1.42	1.00	1.00	1.00
CD	39.27	37.12	24.59	25.13	13.95
DISPERSE	5.78	4.40	5.06	5.13	5.11
FIELD	22.59	26.75	30.47	35.24	38.72
FLR	6.65	5.76	4.32	2.83	1.00
FOURTR	1.10	16.33	40.25	41.64	46.22
GE	53.04	53.18	50.00	50.86	45.92
INIT	5.87	5.78	3.46	3.46	3.46
LUD	.1	.12	34.97	31.12	20.44
MAIN	5.11	7.32	6.85	7.10	6.72
MAMOCO	1.07	1.10	3.52	3.59	4.28
MATMUL	58.91	58.91	58.91	58.91	58.91
MATTRP	6.88	5.48	5.48	3.52	3.52
PAPUAL	1.31	1.31	1.31	1.31	7.87
TWOWAY	2.90	2.69	7.55	16.02	17.46
MIN.	.1	.12	1.00	1.00	1.00
AVG.	16.51	17.70	21.74	22.13	21.42
MED.	5.87	5.78	7.55	16.02	13.95
MAX.	58.91	58.91	58.91	58.91	58.91

loops reference similar pages or not. This increase in the number of page faults will disappear, however, as slightly more memory is alloted.

In Table 10 f is the page fault function of the untransformed programs and f_t is the page fault function for the transformed programs. m is the memory allotment in pages. The table shows the ratio of f/f_t at memory allotments of 4, 5, 6, 7, and 8 pages. We note that the average improvements at these page allotments are 16.51, 17.70, 21.74, 22.13, and 21.42 respectively. The average of these averages is 19.9. With a memory allotment of 4 pages, the factor of improvement is greater than 36 for 4 programs, between 22 and 30 for two programs, between 5 and 7 for 5 programs, between 2 and 4 for 2 programs, and less than 2 for 4 programs. With 6 pages, the factor of improvement is greater than 30 for 7 programs, around 25 for 1 program, between 4 and 8 for 5 programs, between 3 and 4 for 2 programs and less than 2 for 2 programs. Finally with 8 pages, the factor of improvement is greater than 34 for 6 programs, between 13 and 20 for 3 programs, between 4 and 8 for 4 programs, between 3 and 4 for 2 programs, and no improvement for 2 programs. We note that only one transformed program produced more page faults than the original program at $m = 4$ and 5. This happened in program LUD because we used loop fusion while transforming its nonbasic into basic π-block. For memory allotments greater than 6, however, the transformed program produces fewer page faults.

We now use the second approach to measure the achieved improvements. Namely we will compare the amount of memory required in the untransformed programs machine to the memory required in the transformed programs machine when both operate at the same level of performance.

Examining the page fault curves in the Appendix one notes that for both the transformed and untransformed programs the curves are monotonically decreasing. For the untransformed programs, the drop in page faults as memory allotment increases is rather gradual in most parts of the curves. In some instances fast drops in faults do occur. The relative magnitudes of such drops are rather small when they happen at small memory allotment. If large sudden drops in faults are observed they usually occur at large memory allotments. Eventually the curves will be asymptotic to the absolute minimum number of page faults, DP.

The page fault curves of the transformed programs follow a much more consistent pattern. All transformed programs encounter a steep drop in page faults at some memory allotment between 4 and 8 pages. We will call the points in the page fault curves where this happens the knee points. The memory allotment at the knee point is denoted by m_{kt} and the page faults of the transformed program will be $f_t(m_{kt})$. Beyond the knee point, $m > m_{kt}$, the f_t curves approach their asymptotic values with small slopes. Since page fault curves are in general not smooth curves, i.e. the slopes change abruptly, we cannot choose a particular slope to find the exact location of the knee point for each curve. For example saying that the knee point is the point at which the slope of the curve is 135° would not work. Examining the space-time curves for the transformed programs, we noticed that the memory allotment at the absolute minimum space-time cost points can be used to identify the knee points in the page fault curves. If we denote the memory allotment at the minimum space-time cost point of a transformed program by m_{ot}, then we choose to take $m_{kt} = m_{ot}$. Using this method of finding the value of m_{kt} was successful in locating the steep drop regions in the page fault curves. Although m_{ot} and m_{kt} have the same value for each transformed program, we wish to use two symbols to emphasize the distinction between the discussion of mono and multiprogrammed systems.

Let us now restate what we are trying to do. We want to compare the memory allotment needed in the machine executing the untransformed programs to the memory needed in the machine exeuting the transformed programs while both machines operate at the same level of performance. Here we have to decide on the levels of performance to be used in making the comparisons. We will make two sets of comparisons. In the first set we take the performance level achieved by the transformed programs at $m = m_{kt}$ to be the comparison level. In other words we will compare m_{kt} and m_{ckt}, where $f_t(m_{kt}) \leq f(m_{ckt})$(the less than sign is used because the f curves are not continuous curves). Thus, m_{ckt} is the memory allotment needed by the untransformed program to generate no less than $f_t(m_{kt})$ page faults. This type of comparison shows the value of the transformations for each program individually because m_{kt} is in general different for different programs. In the second set of comparisons we are more interested in the improvements across the programs from the OS point of view. In other words, if the machine of the transformed programs has only 4 page frames to be alloted to each of these programs, then it is interesting to know the number of page frames needed by the untransformed programs machine to achieve the same level of performance. We will do this comparison with 4, 6, and 8 page frames.

Table 11 shows the results of the first set of comparisons. We note that m_{kt} ranged from 1 to 8 with an average of 4.53. The median

Table 11. Memory Requirements of Transformed and Original Programs at Similar Performance Levels - the Transformed Programs Knee Points Level.

Program	m_{kt}	m_{kt}/DP	$LP(m_{kt})$	m_{ckt}	m_{ckt}/DP	m_{ckt}/m_{kt}
ADVECT	6	.0265	.229	31	.137	5.17
BASE	5	.0167	.817	38	.127	7.60
BIGEN	2	.0052	.877	5	.013	2.50
CD	3	.1429	.231	11	.537	3.67
DISPERSE	3	.0041	.762	60	.082	20.00
FIELD	8	.1538	.853	18	.346	2.25
FLR	2	.0870	.821	7	.304	3.5
FOURTR	6	.0468	.133	65	.508	10.8
GE	3	.0833	.229	35	.972	11.67
INIT	1	.0041	1.00	64	.267	64
LUD	6	.1667	.231	22	.611	3.67
MAIN	5	.0252	.240	26	.131	5.2
MAMOCO	6	.0068	.360	30	.034	5
MATMUL	3	.0400	.273	34	.453	11.3
MATTRP	2	.0800	1.00	9	.360	4.5
PAPUAL	8	.0056	.989	176	.124	22
TWOWAY	8	.0283	.115	56	.199	7
MIN.	1	.0041	.115	5	.013	2.25
AVG.	4.53	.0542	.541	40.53	.039	11.20
MED.	5	.0283	.360	31	.261	5.2
MAX.	8	.1538	1.00	176	.972	22

is 5. Thus on the average only .0542 of the virtual space of programs needs to be in primary memory to achieve an average LP of .541. The average number of page frames needed in the untransformed programs machine to achieve identical levels of performance is 40.53. This number varies between a minimum of 5 and 176 page frames. The median is 31. On the average 11.20 times more page frames are needed in the untransformed programs machine than are needed in the transformed programs machine. Note that the paged machine running untransformed programs needs on the average .309 of the virtual space of programs in primary memory. This factor of 3.24 reduction of memory needed which was achieved by the introduction of paging to nonpaged systems is surpassed by the amount of reduction of the memory needed in the transformed programs machine from the untransformed programs machine (an average of 11.20 compared to an average of 3.24), where both machines are paged.

Tables 12, 13, and 14 show our second set of comparisons. In these tables we use m_{c4}, m_{c6}, and m_{c8} to denote the memory allotments needed by the untransformed programs to generate no less that $f_t(4)$, $f_t(6)$, and $f_t(8)$ respectively. With 4 page frames the transformed programs machine will have on the average an LP of .382 with a median of .244. In Table 12 we note that the untransformed programs machine need on the average 29.35 page frames to achieve the same level of performance with a median of 12.00 page frames. Thus the transformed programs machine achieves an average factor of 7.34 reduction in the required memory to achieve this level of performance (the median is 3.00). Note that on the average, the untransformed programs machine is achieving a factor of 26.25 saving in primary memory compared to an unpaged machine. The transformed programs machine is achieving a factor of 74.40.

Table 12. Memory Requirements of Transformed and Original Programs at Similar Performance Levels - the Transformed Programs 4 Pages Level.

Program	$LP_t(4)$	m_{c4}	$m_{c4}/4$	DP/4	DP/m_{c4}
ADVECT	.022	14	3.50	56.50	16.14
BASE	.444	38	9.50	75.00	7.89
BIGEN	1.00	6	1.50	96.25	64.17
CD	.244	11	2.75	5.25	1.91
DISPERSE	.763	60	15.00	183.50	12.23
FIELD	.369	11	2.75	13.00	4.73
FLR	.885	7	1.75	5.75	3.29
FOURTR	.003	5	1.25	32.00	25.6
GE	.234	35	8.75	9.00	1.03
INIT	1.00	64	16.00	61.25	3.83
LUD	.0001	1	.25	9.00	36
MAIN	.071	14	3.50	49.50	14.14
MAMOCO	.0101	4	1.00	218.75	218.75
MATMUL	.272	34	8.50	18.75	2.21
MATTRP	1.00	9	2.25	6.25	2.78
PAPUAL	.165	174	43.50	354.50	8.15
TWOWAY	.0102	12	3.00	70.50	23.50
MIN.	.0001	4.00	1.00	5.25	1.03
AVG.	.382	29.35	7.34	74.40	26.25
MED.	.244	12.00	3.00	49.50	8.15
MAX.	1.00	174	43.50	354.50	218.75

Table 13. Memory Requirements of Transformed and Original Programs at Similar Performance Levels - the Transformed Programs 6 Pages Level.

Program	$LP_t(6)$	m_{c6}	$m_{c6}/6$	DP/6	DP/m_{c6}
ADVECT	.229	31	5.17	37.67	7.29
BASE	.833	38	6.33	50.00	7.89
BIGEN	1.00	6	1.00	64.17	64.17
CD	.280	11	1.83	3.50	1.97
DISPERSE	.885	64	10.67	122.33	11.47
FIELD	.571	14	2.33	8.67	3.71
FLR	.962	7	1.17	3.83	3.29
FOURTR	.133	65	10.83	21.33	1.97
GE	.246	35	5.83	6.00	1.03
INIT	1.00	64	10.67	40.83	3.83
LUD	.237	22	3.67	6.00	1.64
MAIN	.281	28	4.67	33.00	7.07
MAMOCO	.367	30	5.00	145.83	29.17
MATMUL	.272	34	5.67	12.50	2.21
MATTRP	1.00	9	1.50	4.17	2.78
PAPUAL	.165	174	29.00	236.33	8.15
TWOWAY	.039	17	2.83	47.00	16.59
MIN.	.039	6.00	1.00	3.50	1.03
AVG.	.499	38.18	6.36	49.52	10.25
MED.	.281	30.00	5.00	33.00	3.83
MAX.	1.00	174	29.00	236.33	64.17

Table 14. Memory Requirements of the Transformed and Original Programs at Similar Performance Levels - the Transformed Programs 8 Pages Level.

Program	$LP_t(8)$	m_{c8}	$m_{c8}/8$	DP/8	DP/m_{c8}
ADVECT	.245	31	3.88	28.25	7.29
BASE	.855	38	4.75	37.50	7.89
BIGEN	1.00	8	1.00	48.13	48.13
CD	.349	11	1.38	2.63	1.91
DISPERSE	.893	64	8.00	91.75	11.47
FIELD	.855	19	2.38	6.50	2.74
FLR	1.00	8	1.00	2.88	2.88
FOURTR	.155	65	8.13	16.00	1.97
GE	.275	35	4.38	4.50	1.03
INIT	1.00	64	8.00	30.63	3.83
LUD	.234	22	2.75	4.50	1.64
MAIN	.313	38	4.75	24.75	5.21
MAMOCO	.469	30	3.75	109.38	29.17
MATMUL	.272	34	4.25	9.38	2.21
MATTRP	1.00	9	1.13	3.13	2.78
PAPUAL	.99	176	22	177.25	8.06
TWOWAY	.115	59	7.38	35.25	4.78
MIN.	.115	8	1.00	2.63	1.03
AVG.	.592	41.82	5.23	37.20	8.47
MED.	.469	34.00	4.25	28.25	3.83
MAX.	1.00	176	8.73	177.25	48.73

Table 13 shows similar data when the transformed programs machine allots 6 page frames to all programs. The average LP is .499 (the median is .281). The untransformed programs machine needs on the average 38.18 pages to achieve this level of performance which is an average factor of 6.36 more than the memory needed by the transformed programs machine. On the average, the untransformed programs machine is achieving a factor of 10.25 savings in primary memory (compared to a nonpaged machine) while the transformed programs machine is achieving a factor of 49.52.

Table 14 shows the data when 8 page frames are alloted to all the transformed programs. The average LP is .592 (.469 median). The average memory needed by the untransformed programs is 41.82, which is an average factor of 5.23 more page-frames than 8.

Thus from Tables 10 through 14 it is clear that with few page frames (4 to 8) the transformed programs have a much lower rate of page faulting (on the average a factor of 19.9 lower). To achieve similar levels of page faulting, the untransformed programs need on the average a factor of 5.23 to 7.34 more memory (on the average 29.35 to 41.82 page frames compared with 4-8 page frames for the transformed programs).

4.2.2 The Space-Time Cost vs. Memory Allotment Results

As discussed in Chapter 2, the throughput of a multiprogrammed machine is inversely proportional to the average space-time cost of execution of programs. Thus the concern here is to reduce the space-time cost of programs. Moreover, one would like to reduce the amount of memory alloted to each program because this will improve the degree of multiprogramming.

In this section we compare the space-time cost of executing untransformed and transformed programs. Here we assume that the OS uses the local LRU replacement algorithm and a fixed memory allotment policy. In other words, when a program is executed it is assigned a fixed amount of memory. When this program generates a page fault the OS will replace, if necessary, one of the pages of the same program. In later sections of this chapter we will discuss the implications of our results when the OS uses different memory management strategies.

Traditionally people have used the number of memory references made by a program to measure the time spent by the CPU to execute the program. If we denote this number by R then the space time cost of executing a program under our assumptions is given by:

$$\text{Space-Time Cost} = m * (R + f(m) * T) \qquad 4.1$$

where m is the number of page frames alloted to the program, f(m) is the number of page faults and T is the average page fault service time (in memory references). With the same m the space-time cost of the transformed version of the program is given by:

$$(\text{Space-Time Cost})_t = m * (R + f_t(m) * T) \qquad 4.2$$

We note that equations 4.1 and 4.2 have a common term m*R. If we plot the curves representing these equations (versus memory allotment) then this term is a common, bias to both curves. The bias term of the space-time cost of a program is independent of its degree of locality. The locality of the program affects only the non-bias term. Thus to compare the improvement in the locality of programs one needs to compare only the non-biased space-time costs of the original and transformed programs. This is in some way analogous to measuring the voltage gain of an amplifier by the ratio of the AC output voltage to the AC input voltage.

In the Appendix we show the space-time cost curves for our programs after removing the bias terms. These curves are also independent of the value of T, the page fault service time. We have normalized these curves by making T equal to one unit time, i.e., one unit of the space-time cost is equal to a page frame-page fault service time. Thus the curves represent $m*f(m)$ and $m*f_t(m)$ for the original and transformed programs respectively. We denote these two functions by $ST(m)$ and $ST_t(m)$. Note that the difference between $ST(m_1)$ and $ST_t(m_2)$ is equal to the difference between the total values of the space-time costs when $m_1 = m_2$. However, if $m_1 > m_2$ then $ST(m_1) - ST_t(m_2)$ is less than the difference between the total values of the space-time cost. This is because the bias term, $m*R$, increases as m is increased and hence it will be greater for $ST(m_1)$ than for $ST_t(m_2)$. Thus the comparisons which we will make shortly are on the conservative side (we will be comparing $ST(m_1)$ and $ST_t(m_2)$ with either $m_1=m_2$ or $m_1 > m_2$). In other words our results would have been better if we plotted the total values of the space-time cost functions. In the rest of this thesis, unless otherwise specified, we use the term space-time cost to mean the total space-time cost minus the bias term. Thus for the original programs the space-time cost will be given by the $ST(m)$ function and for the transformed programs by the $ST_t(m)$ function.

Both the ST and ST_t curves have absolute minimums. We will use M_o to denote the memory allotment at the minimum point of the ST curve.

Similarly we use M_{ot} to denote the memory allotment at the minimum point of the ST_t curve. Table 15 shows M_o's for all our programs. We note that M_o ranges between 1 and 67 with an average of 24.8 and a median of 24. There are 6 programs with $M_o < 10$, 7 programs with $M_o > 30$, and 4 programs with $10 \leq M_o \leq 30$. In each of these three sets of programs M_o is spread over the range of the set. In the first range M_o takes the values 1, 1, 6, 6, 8, and 9. In the second set the values are 13, 20, 24, and 28. In the third set the values are 31, 32, 36, 39, 41, 60, and 67.

Thus the first important observation we make is that M_o's of the original programs are well scattered over a wide range.

Another important observation which we make is that _the ST curves are not well behaved for $m < M_o$_ (see the Appendix). For some parts of this memory range ST increases with m for others it decreases. Moreover, often sudden jumps in the value of ST are encountered. In other words the ST curves wiggle, going up and down for $m < M_o$. For $m \geq M_o$ the ST functions are rather linearly increasing with m. Since M_o is scattered over a wide range, it is impossible to choose a narrow band of memory allotment in which all programs will run efficiently, i.e. with ST values close to $ST(M_o)$.

In Table 15 we also show the ratios M_o/DP and $ST(M_o)/DP^2$, where DP is the number of distinct pages referenced. These are intended to give a feeling for the potential advantage that paged virtual memory machines have over non-virtual memory machines. If a program is alloted a number of page frames equal to its M_o, then on the average it will be using only .303 of the memory it needs in a non-virtual memory machine and its space-time cost will be only .388 of the cost in the non-virtual memory machine.

Table 15. Characteristics of the Minimum Space-Time Cost Points of the Original Programs.

Program	M_o	M_o/DP	$ST(M_o)/DP^2$
ADVECT	32	.1416	.2218
BASE	39	.1300	.1300
BIGEN	6	.0156	.0156
CD	13	.6190	.6485
DISPERSE	1	.0013	.0186
FIELD	20	.3846	.4215
FLR	8	.3478	.3478
FOURTR	67	.5234	.6583
GE	36	1.000	1.000
INIT	6	.0244	.0846
LUD	24	.6667	.6587
MAIN	28	.1414	.4900
MAMOCO	31	.0354	.0357
MATMUL	41	.5467	.5467
MATTRP	9	.3600	.3600
PAPUAL	1	.0007	.0167
TWOWAY	60	.2127	.9431
MIN.	1	.0007	.0156
AVG.	24.8	.303	.388
MAX.	67	1.0	1.0
MED.	24	.2127	.3600

The space-time cost curves of the transformed programs have a much better behavior. <u>The minimum points in the ST_t curves occur at memory allotments which fall in a much narrower band.</u> Table 16 shows the M_{ot}'s of our programs. We note that all the transformed programs have $1 \le M_{ot} \le 8$. There are 3 programs with $M_{ot} = 8$, 4 with $M_{ot} = 6$, 2 with $M_{ot} = 5$, 4 with $M_{ot} = 3$, 3 with $M_{ot} = 2$, and one program with $M_{ot} = 1$. The average M_{ot} is 4.53 and the median is 5. The implications of the difference in the range of M_o and M_{ot} and in the behavior of the ST and ST_t curves will be discussed shortly.

In Table 16 we also show M_{ot}/DP and $ST_t(M_{ot})/DP^2$. On the average, when a transformed program is alloted a number of page frames equal to its M_{ot} then it will be using .0542 of its virtual space (which is the same as the virtual space of the untransformed program) and it will be costing only .1822 its cost on a non-virtual memory machine.

Table 17 compares the optimum ST and ST_t points. On the average an untransformed program needs 5.66 more primary memory to achieve its minimum space-time cost. Moreover, the minimum cost of an untransformed program is on the average 4.04 more than the minimum cost of the transformed programs. Note that if the untransformed program was alloted M_{ot} page frames then it will cost (on the average) 29.84 more than the transformed program cost.

Although comparing the optimum ST and ST_t points does serve the purpose of showing the effectiveness of our transformations in improving the behavior of programs and reducing their execution costs, it is still more interesting to make comparisons under more practical assumptions. The point is that an OS has no means of determining the values of M_o or M_{ot} and hence we cannot expect an untransformed program to run with M_o

Table 16. Characteristics of the Minimum Space-Time Cost Points of the Transformed Programs.

Program	M_{ot}	M_{ot}/DP	$ST_t(M_{ot})/DP^2$
ADVECT	6	.0265	.1157
BASE	5	.0167	.0203
BIGEN	2	.0052	.0059
CD	3	.1429	.6190
DISPERSE	3	.0041	.0054
FIELD	8	.1538	.1804
FLR	2	.0870	.1059
FOURTR	6	.0468	.5315
GE	3	.0833	.3634
INIT	1	.0041	.0041
LUD	6	.1667	.722
MAIN	5	.0252	.1052
MAMOCO	6	.0068	.0190
MATMUL	3	.0400	.1467
MATTRP	2	.0800	.080
PAPUAL	8	.0056	.0057
TWOWAY	8	.0283	.2467
MIN.	1	.0041	.0041
AVG.	4.53	.0542	.1822
MAX.	8	.1667	.722
MED.	5	.0283	.1059

Table 17. Comparing the Minimum Space-Time Cost Points of the Original and Transformed Programs.

Program	M_o/M_{ot}	$ST(M_{ot})/ST_t(M_{ot})$	$ST(M_o)/ST_t(M_{ot})$
ADVECT	5.3	36.18	1.917
BASE	7.8	54.00	6.376
BIGEN	3	40.49	2.361
CD	4.3	46.32	1.047
DISPERSE	.3	9.41	3.477
FIELD	2.5	38.72	2.336
FLR	4	7.5	3.286
FOURTR	11.17	40.25	1.872
GE	12	53.73	2.751
INIT	6	42.29	20.74
LUD	4	34.97	.949
MAIN	5.6	7.32	4.656
MAMOCO	5.2	3.52	1.881
MATMUL	13.7	58.97	3.727
MATTRP	4.5	7.72	4.5
PAPUAL	.125	7.87	2.923
TWOWAY	7.5	17.46	3.837
MIN.	.125	3.52	1.05
AVG.	5.66	29.84	4.04
MAX.	13.7	58.97	20.74
MED.	5.2	36.78	2.92

page frames or a transformed program to run with M_{ot} page frames. Thus the comparison at the optimum ST and ST_t points is probably only of academic theoretical interest. Although we do not wish at this point to discuss some particular existing OS's, we want to make some comparisons under assumptions which are closer to what happens in the real world.

We will make two sets of comparisons. In the first set we compare ST to ST_t when both the transformed and untransformed programs are allocated similar memory allotments ($4 \leq m \leq 8$). This type of comparison will show us the reduction of the space-time cost which our transformations achieve if the OS uses the policy of alloting a small fixed number of page frames for all programs. In the second set of comparisons we show that on the average, the cost of a transformed program when alloted a number of page frames in the range 4 to 8 is much less (an order of magnitude) than the cost of the untransformed program even if it is alloted a number of page frames from a much larger range ($12 \leq m \leq 48$). Here we will be comparing ST_t at m=4, 6, and 8 to ST at memory allotments in the range $12 \leq m \leq 48$ with an increment of 4 page frames.

Since at a fixed memory allotment, $m = m_a$, we have:

$$ST(m_a)/ST_t(m_a) = m_a*f(m_a)/m_a*f_t(m_a) = f(m_a)/f_t(m_a)$$

then the results of comparing ST to ST_t at similar memory allotments in the range $4 \leq m \leq 8$ are identical to those shown in Table 10. Thus all our previous discussion about the improvements in page faults for this memory range apply directly to the improvements achieved in the space-time cost. Hence, on the average the transformed programs will have 19.9 times less space-time cost than the untransformed programs when all programs are assigned a fixed memory allotment in the range 4 to 8 page frames.

Tables 18, 19, and 20 show our second set of comparisons. In Table 18-a we show for all our programs the ratio $ST(m)/ST_t(4)$, where $12 \leq m \leq 48$. Note that we do not make the comparison for a program at any m which is greater than DP of the program. We observe that for most programs and for most memory allotments we have $ST_t(4) < ST(m)$. This is not true for program ADVECT with $32 \leq m \leq 48$. This is because for ADVECT $M_o = 32$ and $M_{ot} = 6$. When some more memory is given to the transformed version of ADVECT (6 or 8 page frames) ST_t will be less than $ST(m)$ for any $12 \leq m \leq 48$ (Tables 19-a and 20-a). Similar remarks apply to program MAMOCO. In Table 20-a we note that programs CD and LUD are the only two programs for which $ST_t(8)$ is greater than $ST(m)$ for some m, $12 \leq m \leq 48$. The ratio ST/ST_t improves as the transformed versions of these two programs are given less pages. This is because M_{ot} for CD is 3 and for LUD is 6. From Tables 18-a, 19-a, and 20-a it seems that an OS can use the simple rule of allocating 4 pages to the transformed programs with relatively small DP (say less than 100 or 75 page frames) and 8 page frames to those with larger DP's. In this case the transformed programs will (in almost all cases) cost less to execute than the original programs no matter how much memory is assigned to the untransformed programs. (Note that it is not our purpose here to determine the exact values of such numbers as 4 page frames for programs with DP < 100, otherwise 8 page frames. More programs and more detailed studies need to be done in order to determine such numbers. However, using statistics available from large collections of Fortran programs and arguments

Table 18-a. $ST(m)/ST_t(4)$, $12 \leq m \leq 48$.

Program \ m	12	16	20	24	28	32	36	40	44	48
ADVECT	8.32	1.40	1.14	1.36	1.10	.28	.32	.35	.39	.42
BASE	10.60	14.12	17.65	21.18	24.71	28.24	31.77	4.44	4.88	5.33
BIGEN	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00	11.00	12.00
CD	.98	.98	1.22	---	---	---	---	---	---	---
DISPERSE	6.34	4.71	5.72	6.87	7.97	8.59	9.63	12.21	11.67	12.68
FIELD	2.21	2.27	2.02	2.38	2.78	3.18	3.57	3.97	4.37	4.77
FLR	2.65	3.54	4.42	---	---	---	---	---	---	---
FOURTR	19.36	23.33	26.65	28.34	29.48	32.77	34.70	36.64	37.85	33.93
GE	12.97	11.21	13.91	15.12	12.34	11.84	2.13	---	---	---
INIT	10.37	13.83	15.82	18.98	12.49	14.27	16.05	17.84	19.62	21.40
LUD	.038	.03	.02	.003	.003	.004	.004	---	---	---
MAIN	3.85	3.40	2.40	2.89	1.73	1.80	1.83	2.00	2.16	2.29
MAMOCO	.28	.37	.45	.53	.57	.08	.09	.10	.11	.12
MATMUL	13.36	17.82	22.27	26.73	31.18	30.55	5.60	5.05	3.00	3.27
MATTRP	3.00	4.00	5.00	6.00	---	---	---	---	---	---
PAPUAL	3.94	5.25	6.56	7.87	9.19	10.50	11.87	13.12	14.44	15.75
TWOWAY	2.25	1.15	1.32	1.33	1.51	1.46	1.60	1.76	1.37	1.48

Table 19-a. $ST(m)/ST_t(6)$, $12 \leq m \leq 48$.

Program \ m	12	16	20	24	28	32	36	40	44	48
ADVECT	56.92	9.55	7.83	9.28	7.51	2.00	2.16	2.40	2.64	2.88
BASE	13.30	17.73	22.15	26.58	31.02	35.45	39.88	5.57	6.13	6.69
BIGEN	2.00	2.67	3.33	4.00	4.67	5.33	6.00	6.67	7.33	8.00
CD	.75	.75	.93	---	---	---	---	---	---	---
DISPERSE	4.90	3.65	4.42	5.31	6.17	6.64	7.44	8.24	9.03	9.81
FIELD	2.29	2.34	2.09	2.46	2.87	3.28	3.69	4.10	4.51	4.92
FLR	1.92	2.56	3.19	---	---	---	---	---	---	---
FOURTR	40.30	48.55	55.46	58.98	61.36	68.19	72.21	76.26	78.78	70.62
GE	9.00	7.78	9.66	10.49	8.57	8.22	1.48	---	---	---
INIT	6.91	9.22	10.54	12.65	8.32	9.51	10.70	11.89	13.08	14.27
LUD	8.38	8.63	6.58	.95	1.08	1.23	1.38	---	---	---
MAIN	10.10	8.92	6.33	7.58	4.55	4.73	4.81	5.25	5.66	6.01
MAMOCO	6.55	8.68	10.80	12.56	13.46	1.94	2.18	2.43	2.67	2.91
MATMUL	8.97	11.88	14.85	17.82	20.79	20.36	3.73	3.37	2.00	2.18
MATTRP	2.00	2.67	3.33	4.00	---	---	---	---	---	---
PAPUAL	2.62	3.50	4.37	5.25	6.12	7.00	7.87	8.75	9.62	10.5
TWOWAY	5.65	2.89	3.31	3.34	3.80	3.66	4.03	4.43	3.45	3.72

Table 20-a. $ST(m)/ST_t(8)$, $12 \leq m \leq 48$.

Program \ m	12	16	20	24	28	32	36	40	44	48
ADVECT	45.65	7.66	6.28	7.45	6.03	1.54	1.73	1.92	2.11	2.31
BASE	10.23	13.64	17.04	20.45	23.86	27.27	30.68	4.29	4.71	5.14
BIGEN	1.50	2.20	2.75	3.30	3.85	4.40	4.95	5.50	6.05	6.60
CD	.69	.69	1.55	---	---	---	---	---	---	---
DISPERSE	3.73	2.77	3.37	4.04	4.69	5.05	5.66	6.27	6.87	7.47
FIELD	2.56	2.62	2.34	2.75	3.21	3.67	4.13	4.59	5.05	5.57
FLR	1.50	2.00	2.5	---	---	---	---	---	---	---
FOURTR	35.03	42.20	48.20	51.27	53.33	59.27	62.77	66.29	68.48	61.38
GE	7.25	6.26	7.78	8.45	6.90	6.62	1.19	---	---	---
INIT	5.19	6.91	7.91	9.49	6.24	7.13	8.03	8.92	9.81	10.70
LUD	6.37	6.56	5.00	.72	.82	.94	1.05	---	---	---
MAIN	8.42	7.43	5.27	6.31	3.79	3.94	4.00	4.37	4.71	5.00
MAMOCO	6.38	8.46	10.52	12.25	13.72	1.89	2.73	2.37	2.60	2.84
MATMUL	6.68	8.91	11.14	13.36	15.59	15.27	2.80	2.53	1.50	1.64
MATTRP	1.50	2.00	2.5	3.00	---	---	---	---	---	---
PAPUAL	11.81	15.75	19.68	23.62	27.56	31.49	35.43	39.37	43.30	47.24
TWOWAY	12.59	6.45	7.38	7.44	8.46	8.16	8.98	9.86	7.68	8.28

about the number of statements in a π-block and the number of operands per statement, we are inclined to believe that our numbers are close to being accurate.)

Tables 18-b, 19-b, and 20-b give some statistics about Tables 18-a, 19-a, and 20-a respectively. With 4 page frames, a transformed program will have on the average between 6.09 and 10.8 times less space time product than the untransformed program when executed with a memory allotment in the range $12 \leq m \leq 48$. The memory reduction is between a factor of 3 and 12 with an average of 7.5 and a median of 7.5. Note that the median of the reduction in the space-time cost ranges between 3.54 and 8.30 with an average of 5.32. The average of the averages of the improvement in the space-time cost is 8.81.

In Table 19-b, with 6 page frames the average improvement in the space-time cost ranges between 8.94 and 12.88 with an average of 11.49. The median of the improvement ranges between 4.42 and 7.78 with an average of 6.31. The reduction in memory ranges between a factor of 2.00 and 8.00 with an average and a median of 5.00.

In Table 20-b the transformed programs are assigned 8 pages. The average reduction of the space-time cost ranges between 8.39 and 13.68 with an average of 11.73. The median of the improvement ranges between 4.54 and 7.45 with an average of 6.03. The reduction in memory ranges between 1.50 and 6.00 with an average and a median of 3.75.

From Tables 18-b, 19-b and 20-b one can say that when transformed programs are executed with memory allotments in the range 4 to 8 pages they have less space-time cost than the untransformed programs by an average factor of 10.68 (this is the average of 8.81, 11.49 and

Table 18-b. Summary of Table 18-a.

m		12	16	20	24	28	32	36	40	44	48
m/4		3	4	5	6	7	8	9	10	11	12
	MIN.	.03	.03	.02	.003	.003	.004	.004	.10	.11	.12
$ST(m)/ST_t(4)$	AVG.	6.09	6.55	7.74	9.71	10.15	10.80	9.15	8.96	9.45	9.45
	MED.	3.85	3.54	5.00	6.00	7.49	8.30	4.59	4.75	4.63	5.05
	MAX.	19.36	23.33	26.65	28.34	31.18	32.77	34.70	36.64	37.85	33.93

Table 19-b. Summary of Table 19-a.

m		12	16	20	24	28	32	36	40	44	48
m/6		2.00	2.67	3.33	4.00	4.67	5.33	6.00	6.67	7.33	8.00
	MIN.	.75	.75	.93	.95	1.08	1.23	1.38	2.40	2.00	2.18
$ST(m)/ST_t(6)$	AVG.	10.74	8.94	9.95	12.08	12.88	12.68	11.97	11.67	12.08	11.88
	MED.	6.55	7.78	6.33	7.58	6.84	5.99	4.42	5.41	5.90	6.35
	MAX.	56.92	48.55	55.46	58.98	61.36	68.19	72.27	76.26	78.78	70.62

Table 20-b. Summary of Table 20-a.

m		12	16	20	24	28	32	36	40	44	48
m/8		1.50	2.00	2.50	3.00	3.50	4.00	4.50	5.00	5.50	6.00
$ST(m)/ST_t(8)$	MIN.	.69	.69	1.55	.72	.82	.94	1.05	1.92	1.50	1.60
	AVG.	9.83	8.39	9.49	11.59	12.68	12.62	12.40	13.02	13.57	13.68
	MED.	6.38	6.56	6.28	7.45	6.57	5.84	4.54	5.05	5.55	6.06
	MAX.	45.65	42.20	48.20	51.27	53.33	59.27	62.77	66.29	68.48	61.38

and 11.73). To achieve this improvement, a transformed program will be executing with a memory which is on the average 5.42 less than the memory alloted to the untransformed program. Thus in a multiprograming system our program transformations can result potentially in an order of magnitude improvement in the throughput with an increase in the degree of multiprograming of more than a factor of 5.

4.3 Measuring the Performance Improvement of Paged Virtual Memory Systems - the Variable Memory Allotment Case

Most existing virtual memory multiprogrammed systems use memory management policies that vary the memory alloted to a program during its execution. Here we choose the working set policy to represent variable memory allotment policies [DENN68]. Other policies are variations and approximations to the working set policy. Our interest is in finding the effect of our transformations on the space-time cost of executing programs under the working set memory management policy.

Several studies have shown that variable memory allotment policies are superior to fixed memory allotment policies like the LRU [CHU72],[COFF72],[DENN75]. The main reason behind the superiority of the variable memory allotment policies is because the main memory requirement of a program may change drastically during its execution. While fixed memory allotment policies assign to a program the same amount of memory during its entire execution time, variable memory allotment policies try to adapt the memory alloted to a program to the changing size of its locality sets.

The working set policy keeps in memory pages referenced during the previous τ references. This set of pages is called the working set and is denoted at time t by $W(t,\tau)$. τ is the window size. The size of the working set at time t is denoted by $w(t,\tau)$. From the results of our experiments reported in Section 4.1, it is obvious that the changes of the sizes of the locality sets of a transformed program are much less than these changes in an untransformed program. Hence it is interesting to see whether the working set policy is any better than the LRU policy for the transformed programs. For untransformed programs, it seems that enough previous work was done to show that variable memory allotment policies are better. More work on these lines seems to be insignificant. Thus our interest is to compare the space-time cost of the transformed programs under the LRU and the working set policy (WS).

Under the LRU policy one can plot the space-time cost as a function of memory allotment. Under the WS policy, however, the memory alloted to a program, i.e. its working set size, $w(t,\tau)$, varies during its execution. Thus in order to make a comparison to the space-time cost under LRU, one needs to calculate the average memory alloted to the program during its execution using the WS policy. With a given window size τ, a program trace of length R references will generate $f_w(\tau)$ page faults. Let $w_i(t_i,\tau)$ be the working set size when the ith page fault occurs, $1 \le i \le f_w(\tau)$. Then if we denote the page fault service time by T, the average memory alloted to the program is given by

$$M(\tau) = \left(\sum_{t=1}^{R} w(t,\tau) + T * \sum_{i=1}^{f_w(\tau)} w_i(t_i,\tau)\right)/(R + T * f_w(\tau))$$

By varying τ one is supposed to get different $M(\tau)$ and $ST_w(\tau)$, the space-time cost under WS, and hence make a plot of $ST_w(\tau)$ versus $M(\tau)$ which can then be compared to the space-time cost curve under LRU.

When collecting data for the $ST_w(\tau)$ curves we found that several programs exhibited anomalous behavior under WS. Recently Franklin, Graham, and Gupta have discovered by experimentation, anomalies with the page fault frequency replacement algorithm [FRAN78]. In the same paper they pointed out that for some reference strings and some τ's the WS policy can also have anomalous behavior. They called these anomalies the parameter (τ)-real memory and real memory-fault rate anomalies. In the paper a short reference string was constructed to illustrate the anomalies with the WS policy.

These are the same anomalies that we found experimentally for some of our transformed programs. Namely, the parameter of the working set policy τ did not have a consistent relation to the average real memory alloted to a program. One expects that the average memory allotment should be a nondecreasing function of τ. In otherwords, given τ_1 and τ_2, if $\tau_2 > \tau_1$ then it is expected that $M(\tau_2) \geq M(\tau_1)$. Similarly one expects the number of page faults generated under WS to be non-increasing with the average alloted memory, i.e. if $M(\tau_2) > M(\tau_1)$ then it is expected to have $f_w(\tau_2) \leq f_w(\tau_1)$. That the WS policy should possess these properties is essential to be able to control the performance of a multiprogrammed system by changing the parameter τ. As it is put in [FRAN78], "...Load control is attempted by varying the paging algorithm parameter. A load control based on an anomalous performance measure may be unstable because a change of given sign in the parameter need not produce changes of corresponding sign in the controlled variable."

For several of our programs we noticed that for some $\tau_2 > \tau_1$ we get $M(\tau_2) < M(\tau_1)$. This is the parameter-average real memory allotment anomaly. Moreover, for $M(\tau_1) > M(\tau_2)$ we noticed that $f_w(\tau_1) > f_w(\tau_2)$. This is the average real memory allotment-page fault anomaly.

To find the average real memory allotment we had to choose a value for T, the page fault service time. For a page size of 64 words, we have chosen to use three different values of T: 32 references, 320 references, and 3200 references (1/2 page size, 5 page sizes, and 50 page sizes). The 3200 value seems to reflect a 64 word page fault service time between disc and primary memory. The 32 references seems to reflect a 64 words page fault service time between an interleaved primary memory and a fast cache memory. Page fault service time between CCD's and primary memory seem to fall between these two extremes [JULI78]. Since our main aim was to compare the space-time cost of the transformed programs under LRU and WS, we have chosen values of τ, the window size, in different ranges and with different increments so as to get $M(\tau)$ in the relevant range of the LRU space-time curve for each program. Generally speaking we used $2 \leq \tau \leq 8$ with an increment of 1 to give us $M(\tau)$ in the range $1 \leq M(\tau) \leq 5$ and we used $\tau \geq 16$ by increments of 8, 32, 64, or 128 to give us $M(\tau) > 5$. The selection of the intial value of τ and its increment was tuned in every program to cover the range of $M(\tau)$ of interest.

Table 21 shows the anomalous behavior of WS which we discovered in 5 of our 17 transformed programs. $M_1(\tau)$, $M_2(\tau)$, and $M_3(\tau)$ are the average alloted memory with the three values of the page transfer time used: 32, 320, and 3200 references respectively. We notice that there is a significant difference between $f_{tw}(\tau_1)$ and $f_{tw}(\tau_2)$. Thus depending on the page fault service time, when the value of τ is increased from τ_1 to τ_2, the reduction in the number of page faults might be big enough

Table 21. Transformed Programs with Anomalous Behavior under WS.

Program	τ_1	τ_2	$f_{tw}(\tau_1)$	$f_{tw}(\tau_2)$	$M1(\tau_1)$	$M1(\tau_2)$	$M2(\tau_1)$	$M2(\tau_2)$	$M3(\tau_1)$	$M3(\tau_2)$
BASE	6	8	681	374	2.33	1.96	2.95	2.37	3.08	2.49
CD	4	5	1904	92	2.79	2.73	2.89	2.82	2.92	3.00
FIELD	6	7	1261	133	3.40	2.86	3.64	3.47	3.67	3.68
MAIN	6	8	6024	1004	3.62	3.06	4.04	3.69	4.11	3.96
MAMOCO	8	16	70938	1878	5.48	4.83	5.59	4.88	5.60	4.91

to make the drop in the space-time integral greater than the drop in time. Thus the average memory allotment will be decreased rather than increased. In general if τ is increased from τ_1 to τ_2, then in order for the anomaly to exist we must have:

$$\left(\left(\sum_{t=1}^{R} w(t,\tau_1) + T * \sum_{i=1}^{f_w(\tau_1)} w_i(t_i,\tau_1)\right)\right/(R + T * f_w(\tau_1)) >$$

$$\left(\left(\sum_{t=1}^{R} w(t,\tau_2) + T * \sum_{i=1}^{f_w(\tau_2)} w_i(t_i,\tau_2)\right)\right)/(R + T * f_w(\tau_2))$$

Thus the existence of the anomaly depends on the program, τ_1, τ_2, and T. We do not see an obvious way of explaining the dependence of the anomaly on each individual one of these factors. The four factors interact to produce the anomaly. In [FRAN78] an argument was presented to support a theory that when the anomaly occurs for a given program, τ_1, and τ_2 then there exists a crossover value of $T = T_c$ such that the anomaly will occur for all $T > T_c$. Our experiments have shown that this theory is not valid. For example in programs CD and FIELD the anomaly occurs for T = 32 and T = 320 but it does not occur for T = 3200.

For all our transformed programs we noted that the anomaly either does not exist or it occurs at values of $M(\tau)$ which are less than M_{ot}, the memory allotment at the minimum space-time point under LRU. We found that for all those programs which are anomaly free there was no difference between the space-time cost under LRU and WS in any memory range and for the three values chosen for T. For the 5 programs which exhibited the anomalous behavior, there was no difference between the space-time cost under LRU and WS for memory allotments greater than M_{ot}. For memory allotments less than M_{ot} the anomaly existed and no comparison can really be made. Note that when we say there was no difference between the cost under LRU and WS we mean that one cannot really draw two different curves to represent the LRU and WS space-time cost functions. In Figures 27, 28, and 29 we show the space-time cost for two programs which have the anomaly (CD and BASE) and for one program which is anomaly free (MATMUL). We will not show curves for any more programs because they do not reveal any additional interesting information.

Because of our observation that the anomaly occured at values of memory allotments less than M_{ot} (which might be interpreted by some people to mean that the anomaly only shows for some programs when they are _thrashing_, whatever the definition of thrashing might be) we did some more experimentation to see whether this is always true. We generated the space-time cost functions under the WS policy for 7 of our untransformed programs, namely ADVECT, BASE, BIGEN, DISPERSE, FOURTR, INIT, and PAPUAL. The anomaly showed in 3 of these programs; INIT, DISPERSE, and FOURTR. For program INIT the anomaly occurred at memory allotments below and above M_o(For INIT M_o=6). For program DISPERSE the anomaly occurred at memory allotments greater than $M_o = 1$. For the FOURTR program the anomaly occurred at memory allotments less than $M_o = 67$. As a matter of fact we did not check whether it also occurs at allotments greater than 67 (Remember that these experiments are very costly because the trace has to be scanned once for every value of τ. We could not find in the literature any algorithm for calculating the _real_ average memory allotments for different τ's in one scan of the trace. Moreover, from

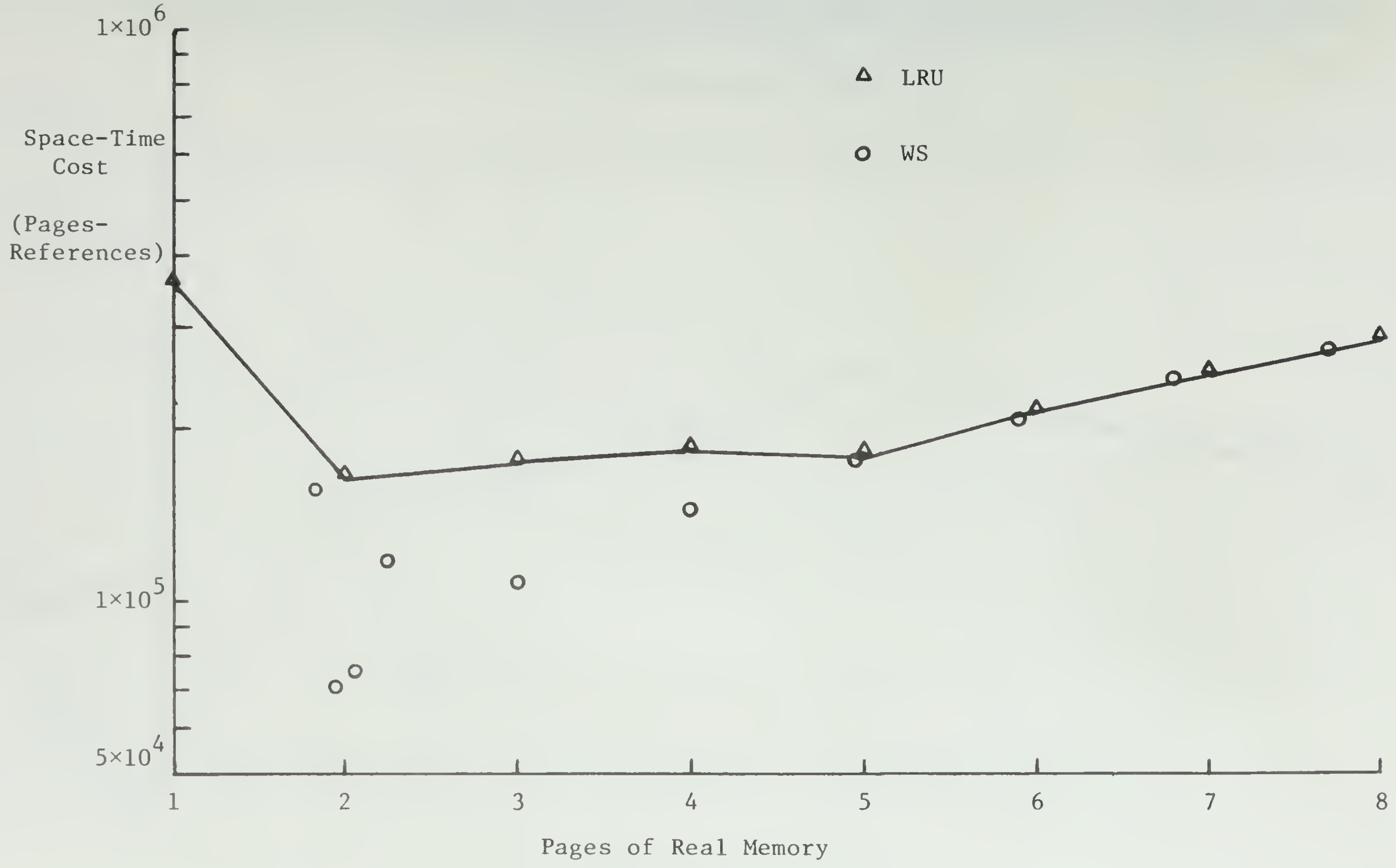


Figure 27-a. The Space-Time Cost of Program BASE (Transformed), T = 32 References

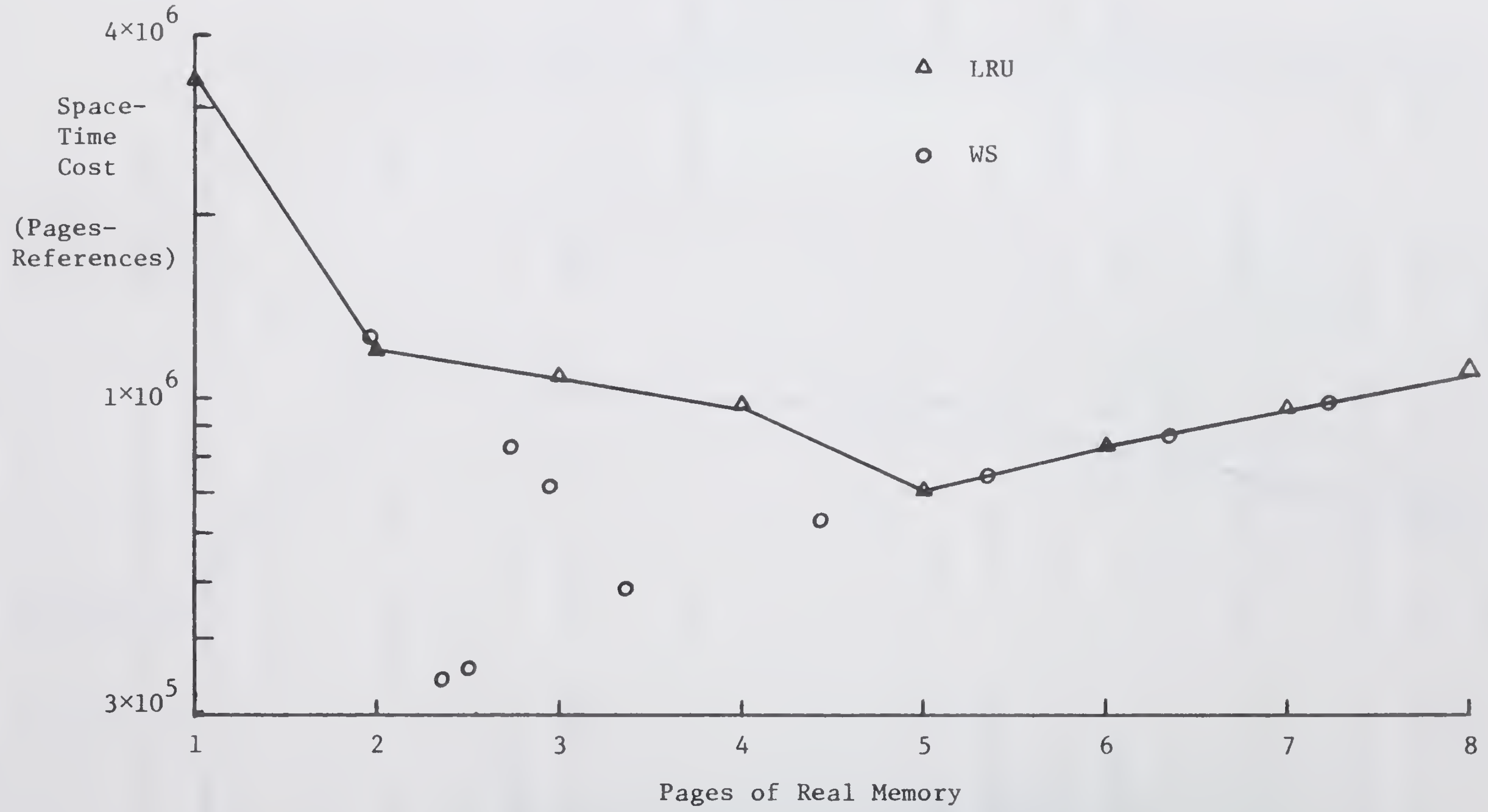


Figure 27-b. The Space-Time Cost of Program BASE (Transformed), T = 320 References

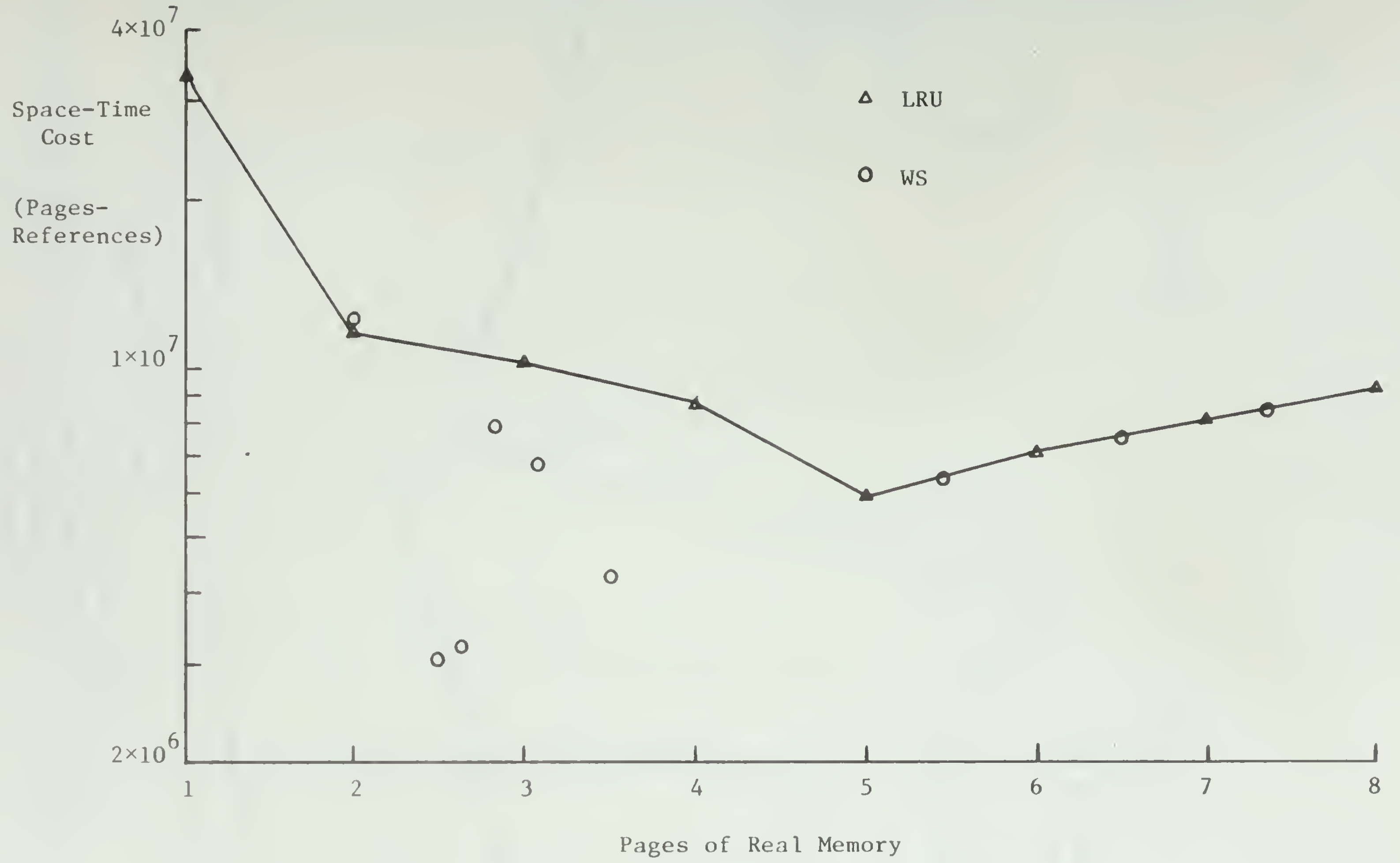


Figure 27-c. The Space-Time Cost of Program BASE (Transformed), T = 3200 References

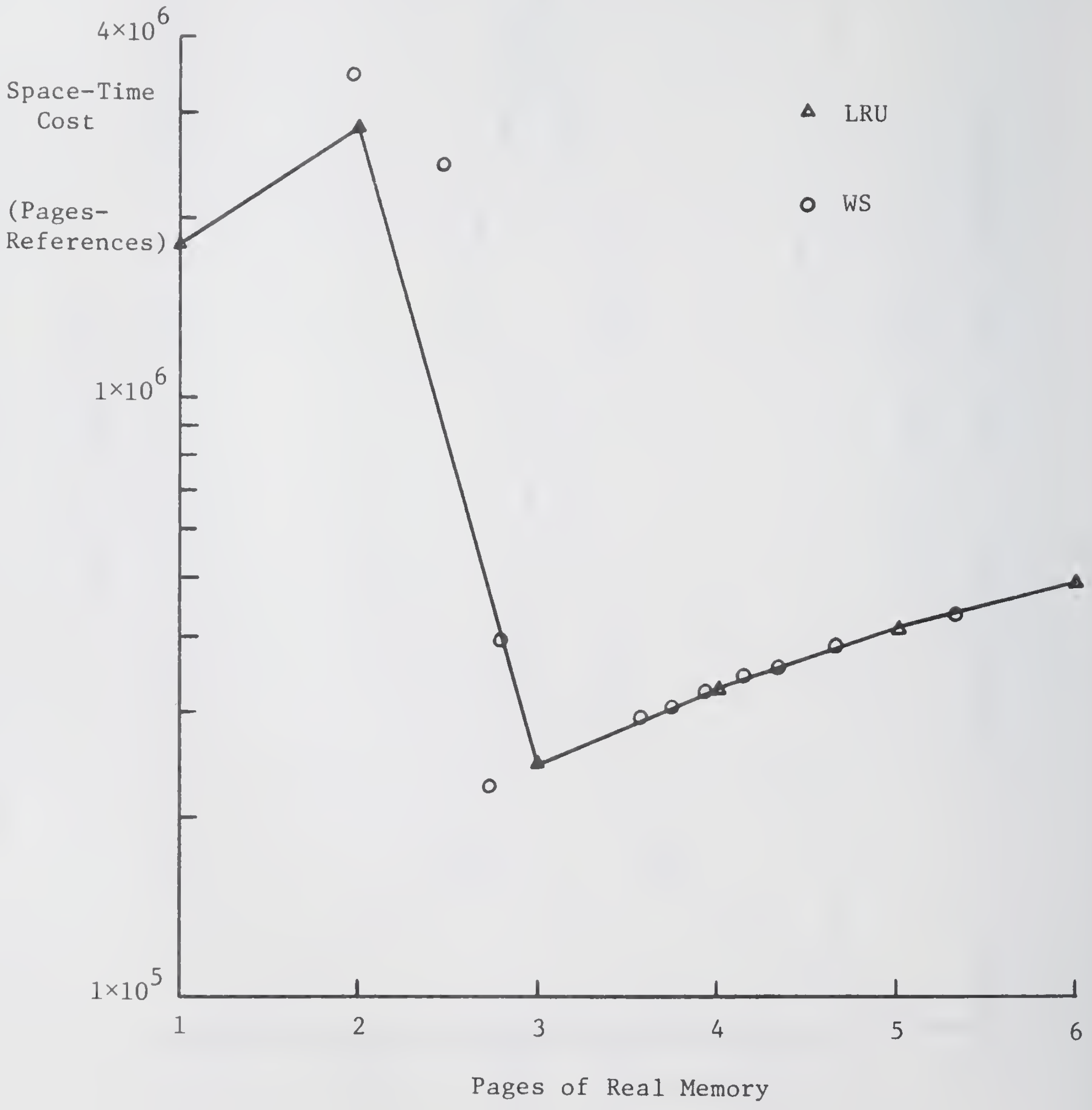


Figure 28-a. The Space-Time Cost of Program CD (Transformed), T = 32 References

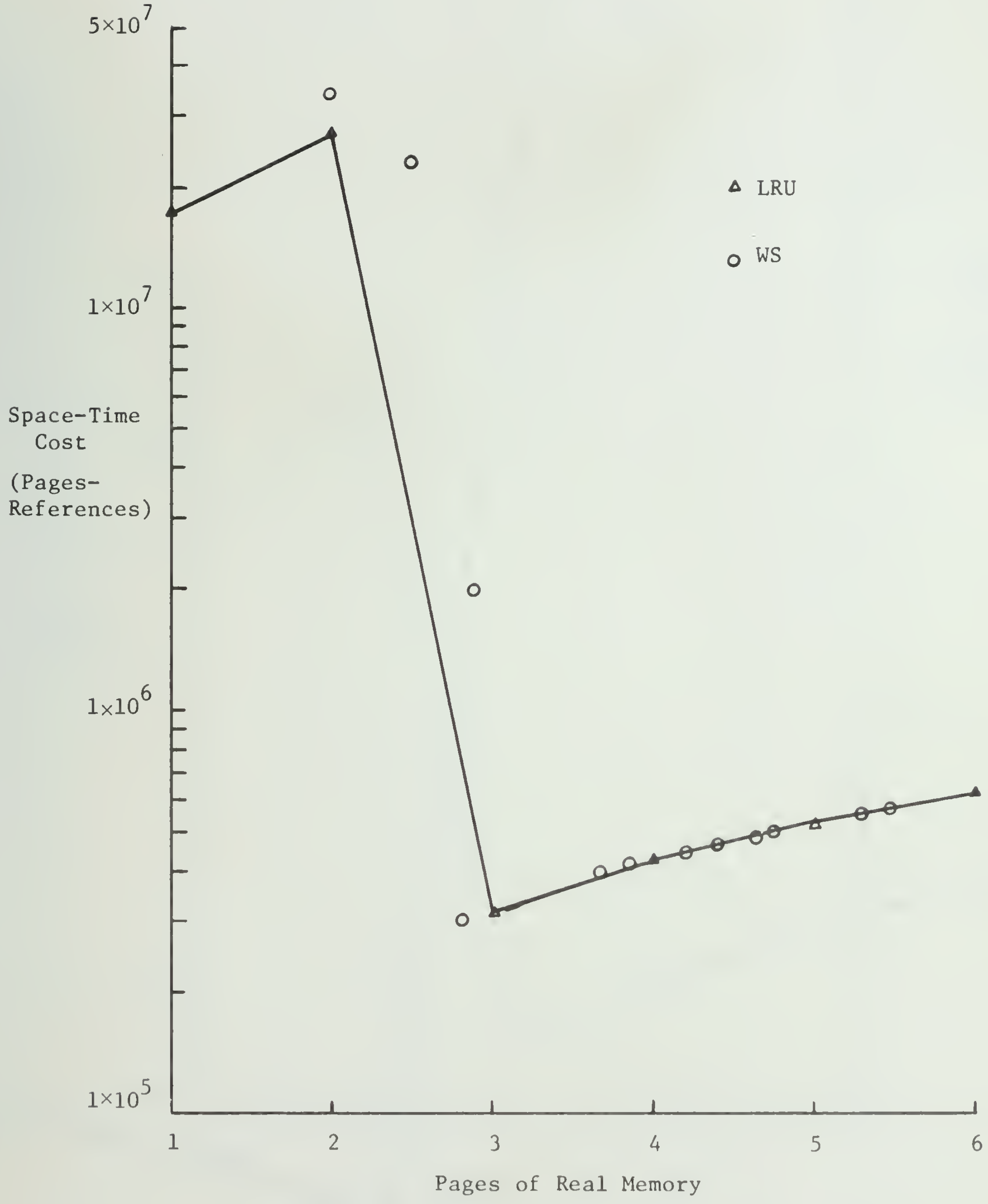


Figure 28-b. The Space-Time Cost of Program CD (Transformed), T = 320 References

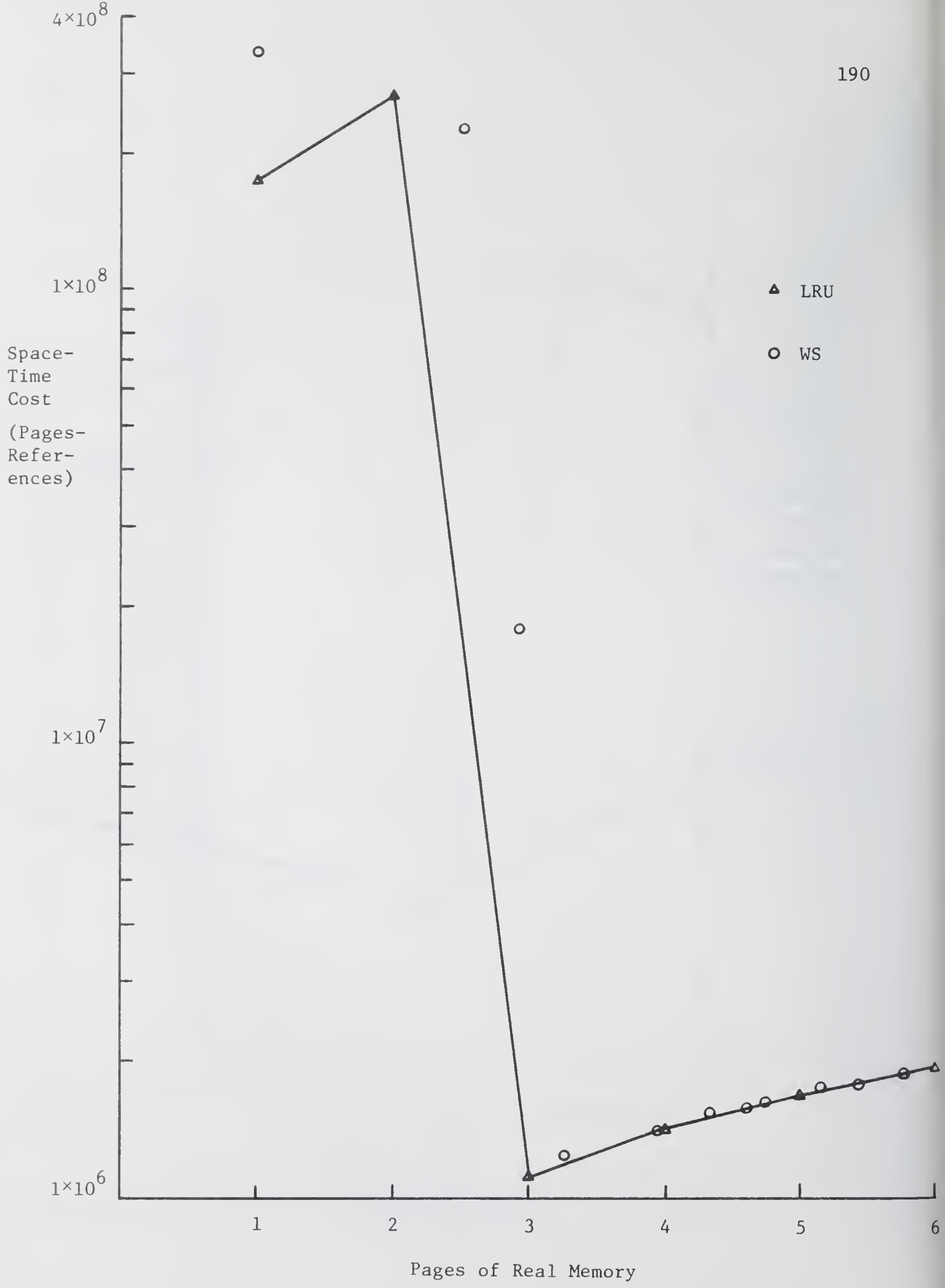


Figure 28-c. The Space-Time Cost of Program CD (Transformed), T = 3200

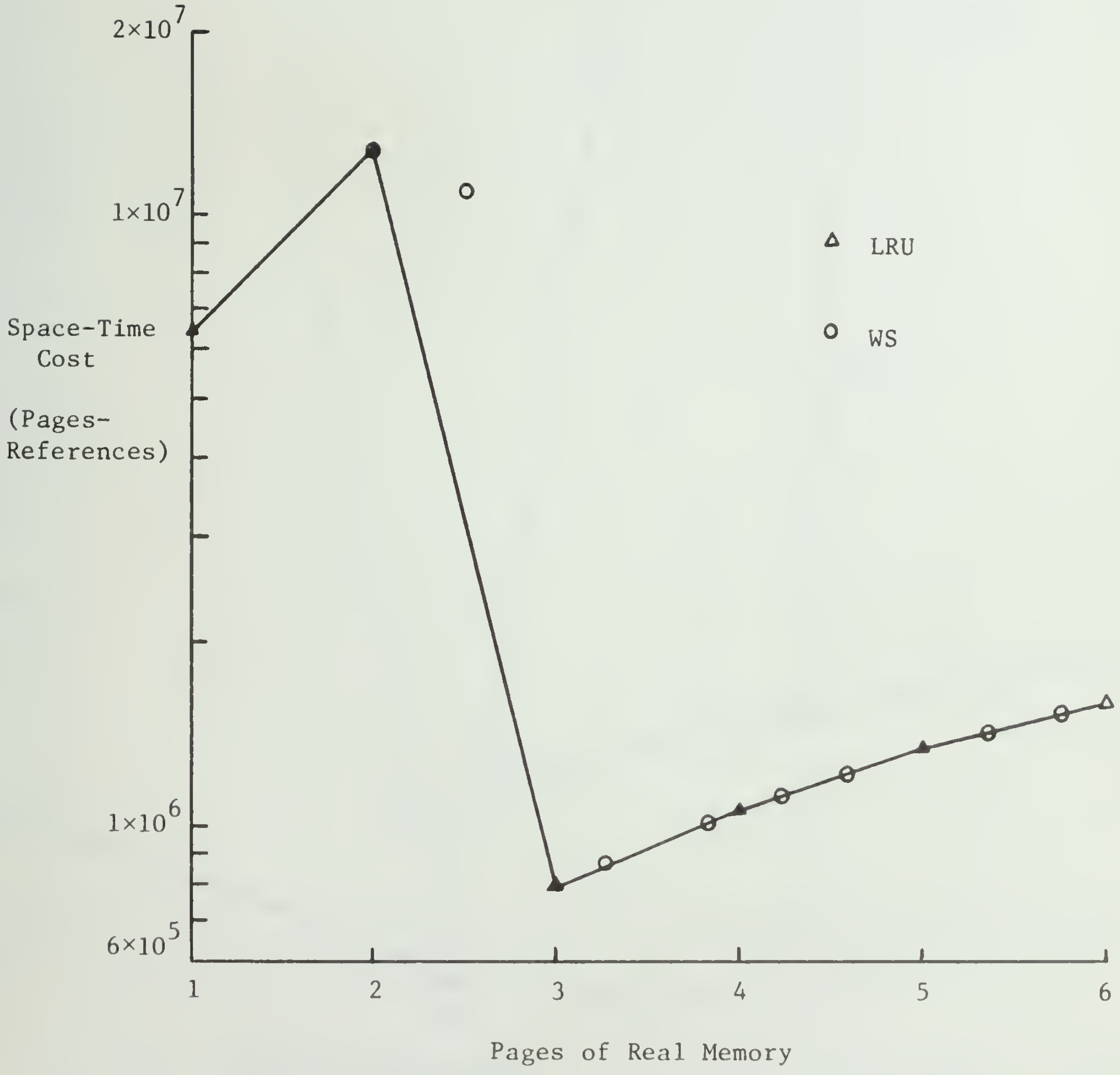


Figure 29-a. The Space-Time Cost of Program MATMUL (Transformed), T = 32

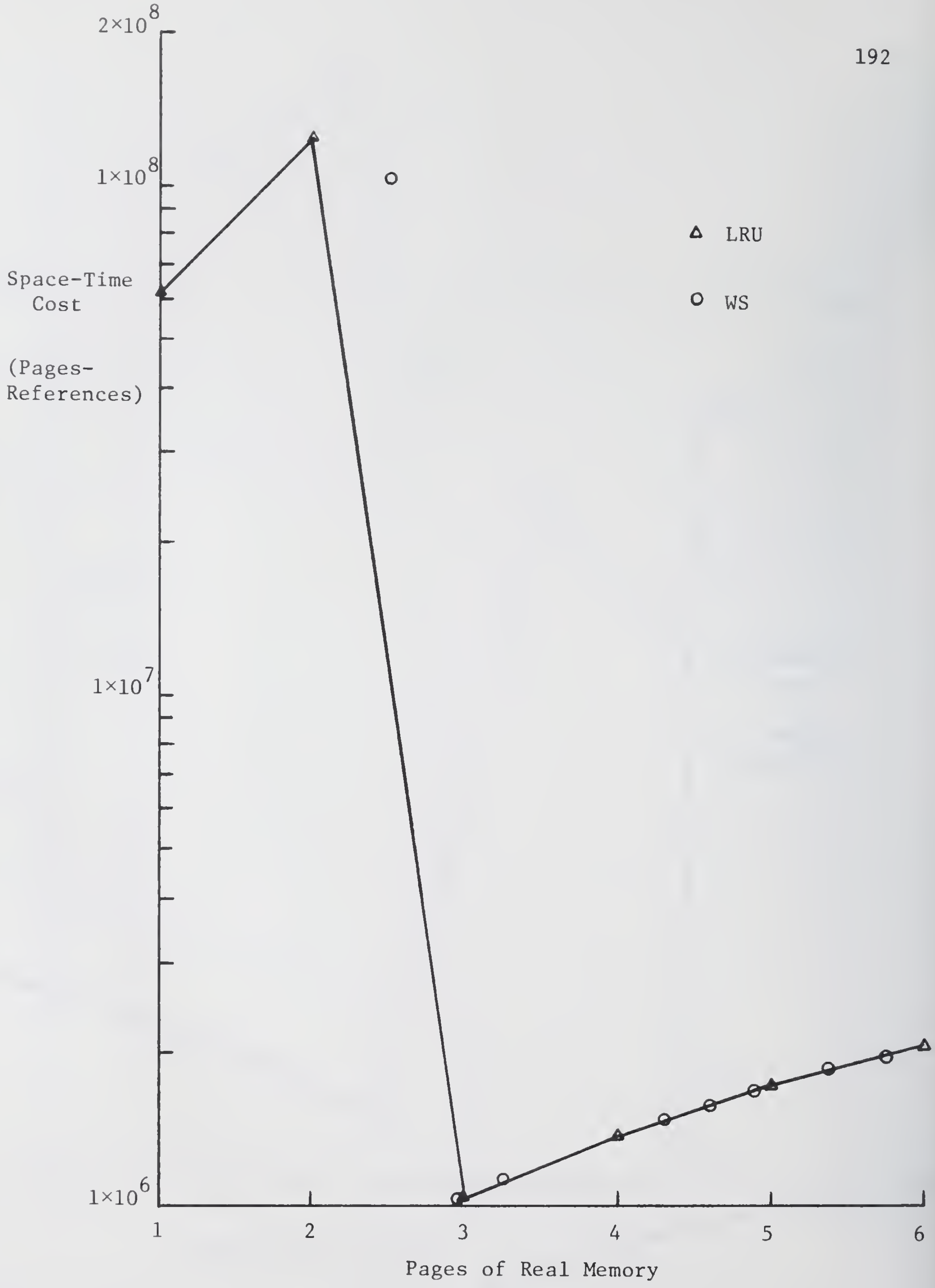


Figure 29-b. The Space-Time Cost of Program MATMUL (Transformed), T = 320

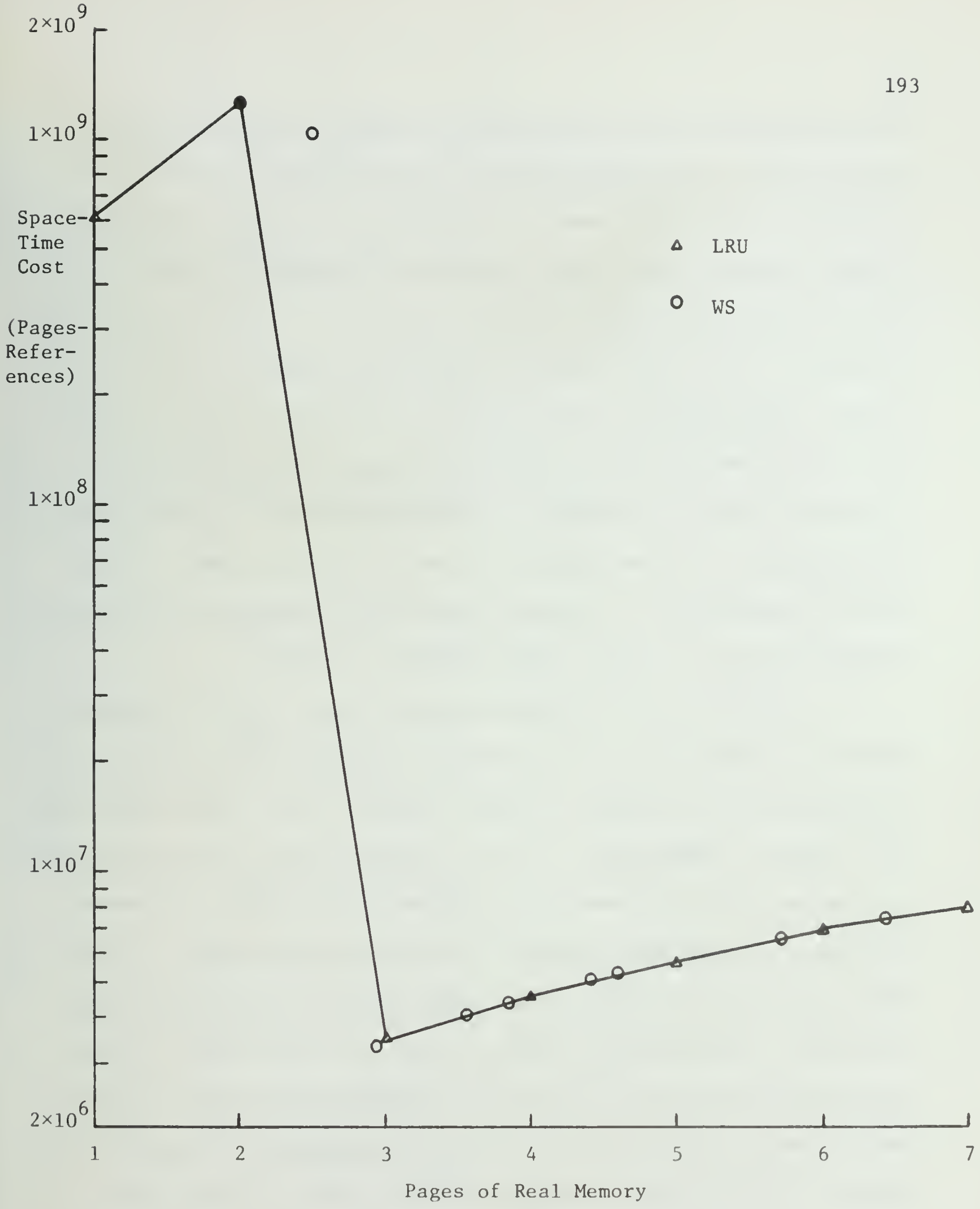


Figure 29-c. The Space-Time Cost of Program MATMUL (Transformed), T = 3200

our own investigation of this matter we reached a conclusion that one needs to save so much information when going through the trace to calculate the real average memory for different τ's, that it is probably cheaper and much simpler to go through the trace several times. To locate anomalies one ideally needs to start at $\tau = 1$ and increase it by increments of 1. This is really a prohibitive expense even for short traces. Most probably, this is the reason why the working set anomaly was not discovered for more than ten years since the introduction of the working set policy [DENN68]. Most probably this is also why nobody else has tried to investigate this anomaly in real programs to date). Table 22 summarizes our findings concerning the anomalies in the untransformed versions of programs INIT, DISPERSE, and FOURTR.

Note that in our previous conclusion there was no difference between the space-time cost of executing a program under the LRU or the WS policies; we are using the average behavior of the program under the WS to make the comparison. In fact it should be clear that the LRU policy is a better policy for transformed programs. If one plots the memory alloted to a transformed program as its execution progresses in real time, the WS curve will have sharp peaks whenever the program changes localities. The LRU curve, however, stays at the same level for the entire execution time of the program. Although the WS sharp peaks are usually short, they can still cause serious problems in a multiprogrammed system. If no free page frames are available when such excessive demand for memory occurs, other programs may be deactivated. In [SMIT76] reducing the seriousness of this problem is approached by making the WS policy more elaborate and introducing a second parameter for the policy.

Table 22. Untransformed Programs with Anomalous Behavior under WS.

Program	τ_1	τ_2	$f_w(\tau_1)$	$f_w(\tau_2)$	$M1(\tau_1)$	$M1(\tau_2)$	$M2(\tau_1)$	$M2(\tau_2)$	$M3(\tau_1)$	$M3(\tau_2)$
INIT	5	6	1417	847	3.80	3.40	3.99	3.51	4.02	3.53
	448	512	437	245	32.38	26.55	38.20	26.72	39.28	26.76
DISPERSE	464	528	948	814	29.37	28.66	32.06	29.65	32.53	29.84
FOURTR	320	384	2788	1903	38.15	37.05	38.62	34.45	38.71	33.84

Smith called his modified WS algorithm the "Damped Working Set Algorithm, DWS." We will not discuss the DWS algorithm and refer the interested reader to [SMIT76]. The point is that _for transformed programs a simple algorithm like the LRU can achieve a level of performance which is as good as the average performance achieved by a more elaborate algorithm (more costly to implement) like the WS which suffers from anomalous behavior for some programs and needs more tuning to avoid the serious problem of the peaks in the memory allotments during the execution of a program_.

4.4 Summary

The preliminary results presented in this chapter show that our transformation techniques are very promising. The transformations have succeeded in making programs behave better, cost less, and they seem to abolish the need for fancy memory management policies. A simple, easy algorithm like the LRU seems to do very well.

The only point that needs clarification is how sensitive are our results and conclusions to the page size which we have chosen, namely 256 bytes or 64 words. This clarification is necessary because most existing virtual memory systems use a page size which is in many cases at least 4 times as large as our pages and more. On the other hand, the sizes of the arrays referenced in the programs may increase. This leads to increasing the loop limits in the program. From talking to some of the sources of our programs we learned that the sizes of the arrays used in their programs might easily grow by a factor of 10. In other cases where we coded programs like the Gaussian Elimination program, we used an array of size 48x48. In many applications, like civil engineering for example, the size of the system of equations solved is much larger than this. Hence we need also to discuss the effect of increasing the array sizes on our results. We spend the rest of this section investigating the effect of varying the page size and array sizes on the page fault and space-time cost curves. The sensitivity of the rest of our results (locality sizes etc.) follow similar lines.

First we discuss transformed programs. Loops in transformed programs follow the ELM model. For a loop that follows the ELM, the critical memory allotment m_o is O(#of different array names). Thus m_o is independent of the array sizes or the page size (as long as the arrays are multi-page arrays). With $m \geq m_o$, the number of page faults f_t is O(K), where K is the number of pages per array. For $m < m_o$, f_t is O(# of words per array). When a loop is alloted $m < m_o$ we will say that the loop is thrashing. Thus, to simplify the discussion, if we assume only one dimensional arrays of size N, then for $m \geq m_o$, f_t is O(N/Z) and for $m < m_o$, f_t is O(N)(see Chapter 2 for more details about these points).

We consider different possibilities. In the first case let us see what happens when the page size Z is increased without increasing the array sizes, or N. In the second case we find out the effects of increasing N without increasing Z. In the last case both N and Z are increased. In all cases we are interested in the programs as long as $N \geq Z$, otherwise their memory requirements are relatively small and they are not of concern to us.

When the array sizes are not changed and the page size is increased, then by extending our previous discussion from the behavior of loops to the behavior of a transformed program, we do not expect M_{ot} to change significantly and for most programs it will not change at all. To see why M_{ot} is not expected to change, let us remember that M_{ot} is the memory allotment at the minimum space-time cost of the program. When Z is increased, the reduction in the number of page faults generated by each loop in the program (when it is not thrashing) is proportional to the reduction in the number of pages spanned by the arrays of the loop. Thus although m_o's of the individual loops are not expected to change, the relative contribution of each loop to the total space-time cost might change. This will happen if relative changes in the number of page faults generated by the loops are not the same. However, since most of the transformed program's time is spent in localities (π-blocks) with five array names or so, the changes in M_{ot}, if they ever occur, will be very little. In other words M_{ot} for any of our programs will always be less than 8 and mostly around 5, irrespective of the page size or the array sizes.

Since as Z is increased the number of pages spanned by each array will decrease, then DP, the number of distinct pages referenced by each program will decrease. Hence the asymptotic value of the page fault curves of both the transformed and untransformed program will drop. Thus the values of $f_t(m)$ for $m \geq M_{ot}$ will decrease. For $m < M_{ot}$, $f_t(m)$ is not expected to change much because the program will be thrashing. This is also true for $f(m)$ of the untransformed program at $m < M_o$. Thus in the memory range $M_{ot} \leq m \leq M_o$ our results will improve. This is because, as mentioned previously, in this range $f_t(m)$ is decreased while $f(m)$ will not drop much. We do expect, however, a drop in M_o which is more appreciable than the change in M_{ot}. Note that for some untransformed programs M_o might not change or changes slightly depending on how well the program is behaving. Thus the general conclusion is that, when the page size is increased then the difference between the $f_t(m)$ and $f(m)$ curves in the region $M_{ot} \le m \le M_o$ is expected to increase (or at least not to decrease) while the width of this region might in general decrease. Similar remarks apply to the $ST(m)$ and $ST_t(m)$ curves. To check the validity of our arguments we have changed the page size (without changing the array sizes) and obtained the page faults and space-time cost data for 4 of our programs: BIGEN, FIELD, MAMOCO, and TWOWAY. The results were in agreement with our expectations. As an example we show in Figures 30 and 31 the faults and space-time cost curves with a page size of 256 words for program MAMOCO and its transformed version. We also show the curves for a page size of 64 words. Note that M_o has dropped from 31 to 17. In BIGEN, with similar changes in page sizes (from 64 to 256) M_o did not change. The untransformed program of BIGEN is much better behaved than MAMOCO. Also, for BIGEN M_{ot} did not change while in MAMOCO M_{ot} increased from 6 to 8 (though $ST_t(6)$ and $ST_t(8)$ for $Z = 64$ are not very different). Note the increase in the improvement in the page faults and space-time cost when Z was increased to 256.

The conclusion we reached in the previous paragraph is really relevant to the validity of our results under the worst possible conditions, namely Z increasing without any increase in the array sizes. A more realistic approach would be to allow both Z and the array sizes to increase. As we have indicated previously, the sizes of arrays can easily grow by a factor of 10 for some of our programs. This is comparable

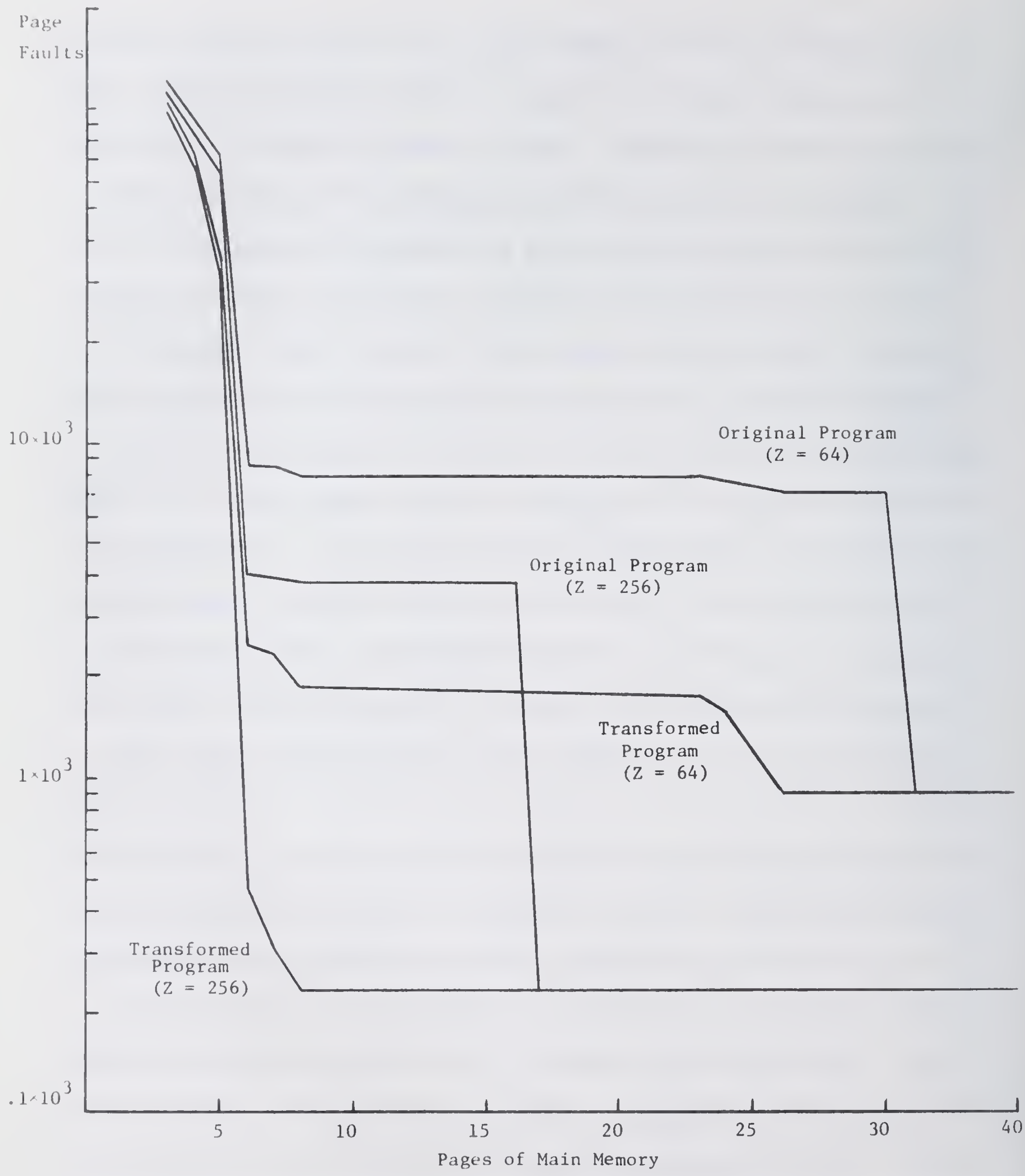


Figure 30. The Page Faults Curves for Program MAMOCO

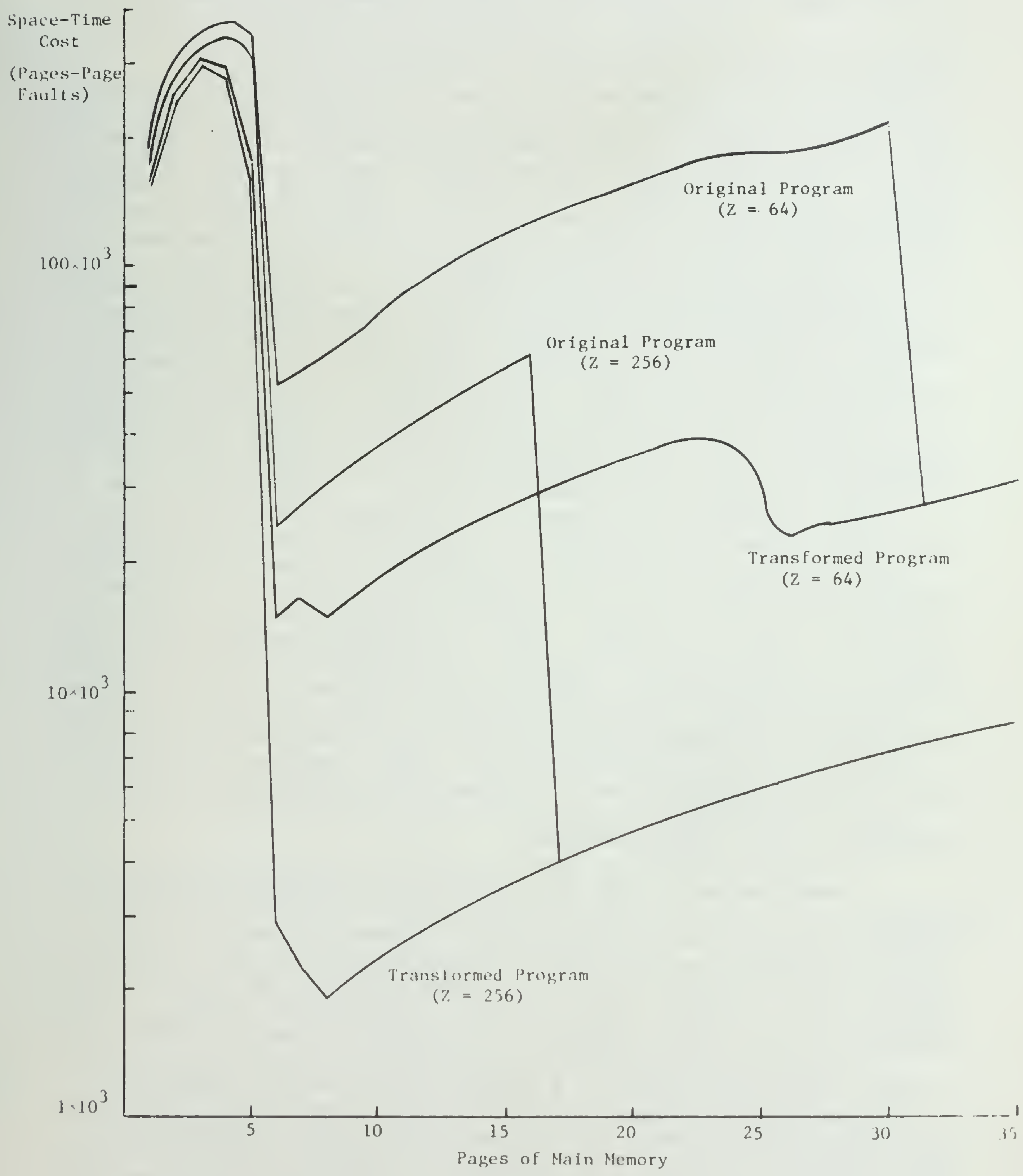


Figure 31. The Space-Time Cost Curves for Program MAMOCO

or even in many cases more than the expected growth of the page size. If the sizes of the arrays grow more than the page size, our results will be improved, and depending on the program, the improvement can be drastic. By an argument similar to what we made previously, M_{ot} is not expected to change much. M_o, however, will increase. Thus, the range of memory allotment which is of concern to us ($M_{ot} \leq m \leq M_o$) will be increased. In this memory range the page faults of the untransformed program will increase in a manner which is roughly proportional to the increase in the number of words per array. The page faults of the transformed program, however, will increase in a manner which is roughly proportional to the increase in the number of pages per array. In other words if we have only one dimensional arrays in a program, the page faults of the untransformed program, $f(m)$, in the range $M_{ot} \leq m \leq M_o$ are in the best case $O(N)$, while f_t is $O(N/Z)$. Thus if the array sizes grow faster than the page size, the region of improvement will increase ($M_{ot} \leq m \leq M_o$) and the degree of improvement will increase. If the page size is increased more than the increase in the array sizes, then we have the situation discussed in the previous paragraph.

What happens if both the page size is increased and the array sizes are increased such that the number of pages per array stays the same? In this case it is easy to see that neither M_o nor M_{ot} will change. Moreover, $f_t(m)$ in the range $M_{ot} \leq m \leq M_o$ will not change. However, $f(m)$ in this range will increase in a manner which is roughly comparable to the increase in the number of words per array. Hence our results will be improved. We believe that this case, where both the array sizes and the page size grow in a comparable way, represents the most realistic situation as far as existing virtual memory machines and the programs which cause problems for these machines are concerned.

To check our conclusions for this latter case we have changed the page size and the array sizes of 6 of our programs such that the number of pages per array stays unchanged. These programs are: CD, FLR, GE, LUD, MATMUL, and MATTRP. Our experimental findings agreed precisely with our expectations. As an example we show in Figures 32 and 33 the curves for program MATMUL. For this matrix multiply program the page sizes are 64 words and 512 words. In both cases each two-dimensional array in the program spanned 25 pages. Thus DP in both cases is 75. For Z = 64 the dimensions of the arrays were 40x40. When we increased Z to 512 we chose the dimensions of the arrays to be 101x101. These dimensions were chosen particularly to be identical to those used by Elshoff for the same program in [ELSH74]. This is because we wanted to compare our results in Figure 32 to the best results obtained by Elshoff when he used all his rules to improve the locality of the same matrix multiplication program. However, this choice of the array dimensions reduces the improvement of our results as Z in changed from 64 to 512. From this point of view it would have been more fair to choose the dimensions to be 110x110. This is because with Z = 64 and array dimensions of 40x40 all points of the 25 pages of each array are referenced (remember we are using the submatrix storage scheme). With Z = 512 and 101x101 arrays only 79.7% of the words in the 25 pages of an array will be referenced. With 110x110 pages 94.5% of the words in the 25 pages of an array will be referenced. Since f(m) for $M_{ot} \leq m \leq M_o$ increases with the number of words referenced while $f_t(m)$ in this range is dependent on

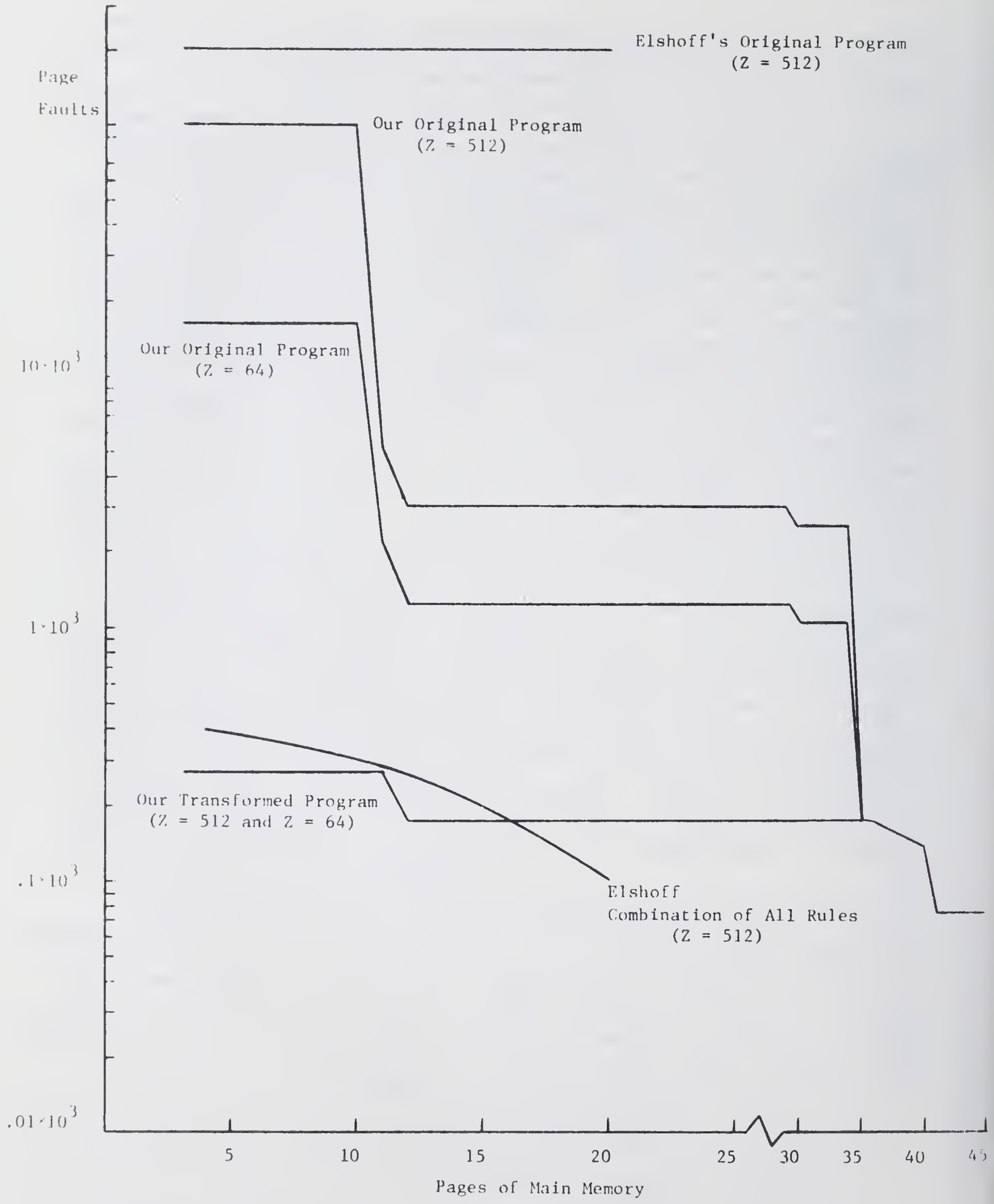


Figure 32. The Page Faults Curves for Program MATMUL

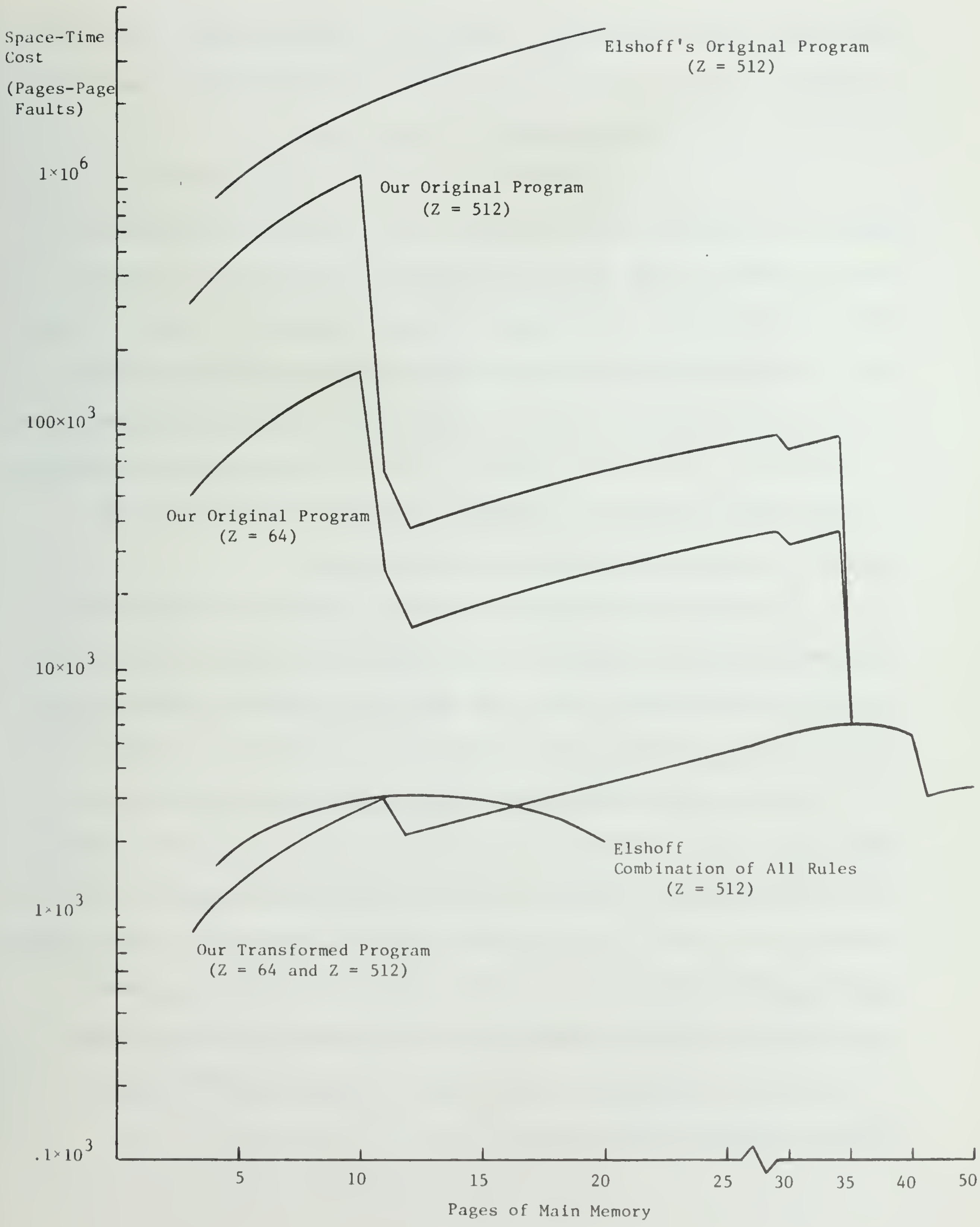


Figure 33. The Space-Time Cost Curves for Program MATMUL

the number of pages referenced, changing the array dimensions from 101x101 to 110x110 would have left $f_t(m)$ unchanged and would have increased f(m) by more than 14.8% (94.5% - 79.7%).

We note that, as expected, the curves of the transformed program are identical for $Z = 64$ and $Z = 512$. For the untransformed program the number of page faults and the space-time cost have increased when Z was increased. The increase in Z is a factor of 8. For $m \leq 10$, f(m) is increased by a factor of 6.17 (for 110x110 arrays the increase in f(m) would be greater than 7.3). Thus, the increases in f(m) and Z are comparable in this memory range. We note that the difference between the f(m) curves decreases as the memory allotment is increased. For $m \geq M_o = 41$, f(m) is independent of the page size.

The data for the Elshoff curves was obtained from [ELSH74](in this paper there is no data for $m > 20$ pages). We observe that our original program produced fewer page faults than Elshoff's original program (for $3 \leq m \leq 10$ the reduction factor is 2 and for $12 \leq m \leq 20$ it is 66.7). We have achieved this improvement simply by storing multi-dimensional arrays using the submatrix storage scheme. Elshoff, however, coded his program in PL1 which stores multi-dimensional arrays by rows. Comparing the curve of our transformed program to the curve of the program using the combination of all Elshoff's rules we note that our automatic transformation techniques (combined with the submatrix storage scheme) are as powerful as Elshoff's rules (for $m \leq 16$ our transformed program produces even fewer page faults).

5. CONCLUSIONS AND EXTENSIONS

We hope that this thesis has been successful in drawing the attention of the computer manufacturers and scientists to the fact that compilers should use special transformations when compiling for virtual memory computers. It is very frustrating to find out that existing compilers do not make any distinction between compiling for a virtual memory machine or for a non-virtual memory machine.

Although in the last decade a tremendous number of papers have been written about virtual memory systems, the behavior and control of these systems are still not well understood. We believe that this is due to the approach taken by many researchers in which programs were treated as black boxes that generate reference strings. More effort needs to be dedicated to studying what is in these boxes, namely the programs themselves. In this thesis we have shown that programs, as written by people, do not behave well in a virtual memory environment. We have also shown that simple compiler transformations can force programs to behave well (and hence be easy to model and manage) and cost less to be executed.

We would like to use the rest of this final chapter to suggest some points for future research. We will discuss three main issues. First, we discuss possible improvements of some of the transformations of Chapter Three. Second, we will raise some questions concerning the implications of our results for the memory hierarchy design problem. Third, we will point out the importance of extending our techniques to non-numeric programs (e.g., Cobol programs).

From all the transformations presented in Chapter Three, the nonbasic to basic π-block transformation seems to be the most costly. The algorithm used in this transformation is simple. However, the number of control instructions executed in the transformed program is increased drastically (for program LUD the increase is almost an order of magnitude--see Table 8). A more elaborate algorithm can be used to apply the page indexing transformation to a nonbasic π-block without first transforming it to a basic π-block. In what follows we illustrate this technique, the non-basic π-block <u>breaking</u> technique, by applying it to Program 16-a of Section 3.5.3.

By definition, the statements of a nonbasic π-block fall at different nest depth levels. The general idea here is to identify the values of the different index variables which cause the recurrence in the π-block and solve the π-block for these values first. Then we will be left with a basic π-block. Consider Program 16-a which is repeated below.

Program 16-a.

```
     DO   S2   I = 1,N
S1   B(I,1) = A(I,1) **.5
     DO   S2   J = 1,N
S2   A(I+1,J) = B(I,J) + C(I,J)
```

By examining the data dependences in this program we find that the recurrence occurs when J = 1 (i.e., the dependence arcs going from S_1 to S_2 and from S_2 to S_1 are due to the fact that J takes the value 1. Thus if J never took the value 1 there will be no recurrence). Hence, this non-basic π-block can be divided into two basic ones as follows:

Program 16-e.

```
         DO     S21   I = 1,N
S11      B(I,1) = A(I,1) **.5
S21      A(I+1,1) = B(I,1) + C(I,1)
         DO     S22   I = 1,N
         DO     S22   J = 2,N
S22      A(I+1,J) = B(I,J) + C(I,J)
```

This program can now be page indexed as follows:

Program 16-f.

```
         DO     S22    I = 1, ⌈N/RZ⌉
         ILB = 1 + (IP-1) *RZ
         IUB = MIN (IP*RZ,N)
         DO     S21   I = ILB,IUB
S11      B(I,1) = A(I,1) **.5
S21      A(I+1,1) = B(I,1) + C(I,1)
         DO     S22   JP = 1, ⌈N/RZ⌉
         JLB = MAX(2,(1 + (JP-1)*RZ))
         JUB = MIM(JP*RZ,N)
         DO     S22   I = ILB,IUB
         DO     S22   J = JLB,JUB
S22      A(I+1,J) = B(I,J) + C(I,J)
```

We have used this concept of breaking nonbasic recurrences in programs CD, GE, and LUD. We obtained the same curves of page faults and space-time cost versus memory allotment as before (for program LUD we got better results here because loop fusion is not used as it was in the nonbasic to basic transformation. M_{ot} is reduced from 6 to 3). Table 23 compares the number of instructions executed when using the recurrence breaking technique and the nonbasic to basic π-block transformation. The advantages of the recurrence breaking technique are obvious. However, more work needs to be done to determine the complexity of this technique and its implementation problems.

Another transformation technique which needs further investigation is one we used in the Fast Fourier Transform program, FOURTR. Basically what we did can be illustrated by the following example.

Program 19-a.

```
        DO S I = 1,N1
        DO S J = I,N2,DELT
    S   A(J) = B(J) + C(J)
```

Table 23.

Comparing the Two Techniques of Transforming Nonbasic π-Blocks.

Program	Number of Instructions Executed		
	Original Program	Nonbasic to Basic π-Block Transformation Used	Recurrence Breaking Transformation Used
CD	234211	2202748	295547
GE	494314	1619039	567741
LUD	507543	2247035	676576

In this program if DELT > N1 then its locality can be improved by transforming it as follows (the mean time between references to the same page will be smaller):

Program 19-b.

```
      DO  S  I = 1, ⌈(N2-1)/DELT⌉
      JLB = 1 + (I-1)*DELT
      JUB = N1 + (I-1)*DELT
      DO  S  J = JLB, JUB
S     A (J) = B(J) + C(J)
```

We have chosen not to discuss this technique in Chapter Three because we did not encounter this situation except once in the programs we examined. More work needs to be done to investigate how important this case is and develop the needed general transformation algorithm.

Before leaving the subject of improving the transformations we want to mention that some of the rules we adopted in some transformations were rather strict. For example, to fuse two NP's we required that their control structure be identical. This rule does not have to be so rigid. Loops of slightly different control structure can be fused if the difference in the control structure is taken care of by appropriate statements (IF statements, for example). Thus the loop fusion transformation might need some tuning.

The second area which has a great potential for further research is investigating the implications of our results for the memory hierarchy design problem. For example, pages of large sizes are currently favored over small pages because of the page fault service time overhead. However, the larger the page the worse the internal fragmentation problem becomes [DENN70]. Currently, with CCD technology, people are building smart (expensive) controllers which reduce the latency time to zero. In [FULL78] and [SITE78], a cheaper approach is suggested which cuts the average latency time to about .1 of the rotation cycle. Thus it seems that the latency problem of the rotating paging devices is going to disappear one way or another. Hence, the page fault service time will be reduced. Since transformed programs have excellent behavior even with small page sizes, then a reconsideration and re-evaluation of the best page size needs to be done. If small page sizes prove to be better, as we expect, then this leads to a considerable reduction in the amount of physical primary memory needed in a machine.

This thesis invites an investigation of another important subject. In the last few years research has been going on at the University of Illinois to design transformations for enhancing the parallelism of ordinary programs to execute efficiently on parallel machines. Not much attention was given to the effect of these transformations on the memory space requirements and I/O activities of programs. The challenging question which we are raising here is how can programs be transformed to run faster on a parallel machine which is supervised by a virtual memory operating system? When transforming programs for vector machines the goal is to maximize the number of operations which can be executed simultaneously. The larger the number of data items which can be processed simultaneously, the higher is the speedup achieved by a vector machine. In other words, parallel and pipelined machines are most effective when they process long vectors. This necessitates that these long vectors will be accessible in main memory. From a paging operating system point of view, however, the goal is to minimize the space-time cost, the primary memory requirements, and the I/O activity of programs. In serial machines the success of virtual memory systems is based on the locality property, i.e., only a small portion (small number of pages) of the data (and code) of a program need to be in main memory at one time. The transformations presented in this thesis are aimed at enhancing this locality property. Thus it seems that our virtual memory enhancement transformations and the parallelism enhancement transformations are at odds. The parallelism transformations assume that all the elements of large arrays will be in main memory, while the virtual memory transformations are designed to make programs execute with as little data in main memory as possible! It is interesting to find out whether some compromise transformations can be designed to achieve both goals: enhancing the parallelism and locality of programs.

Last, but not least, the design of transformations for improving the locality of nonnumeric programs (Cobol programs for example) is another possible area for future research. This is important because the majority of machine cycles in the world are spent on such nonnumerically oriented calculations.

REFERENCES

[ARVI73] Arvid, R. Y. Kain, and E. Sadeh, "On Reference String Generation Processes," Proc. 4th ACM Symp. on Operating Systems Principles, October 1973, pp. 80-87.

[BABO77] Babonneau, J. Y., M. S. Achard, G. Morisset, and M. B. Mounajjed, "Automatic and General Solution to the Adaptation of Programs in a Paging Environment," Proc. 6th ACM Symposium on Operating Systems Principles, November 1977, pp. 109-116.

[BANE76] Banerjee, U., "Data Dependence in Ordinary Programs," Department of Computer Science, University of Illinois at Champaign-Urbana, Report No. 837, November 1976.

[BANE78] Banerjee, U., "Detection of Array Variables in Data Flow Analysis," in Preparation.

[BATS76a] Batson, A. P. and A. W. Madison, "Characteristics of Program Localities," CACM, Vol. 9, No. 5, May 1976, pp. 285-294.

[BATS76b] Batson, A. P. and A. W. Madison, "Measurements of Major Locality Phases in Symbolic Reference Strings," Proc. International Symposium on Computer Performance Modeling, Measurement, and Evaluation, Cambridge, Mass., 1976, pp. 75-84.

[BATS76c] Batson, A. P., "Program Behavior at the Symbolic Level," Computer, November 1976, pp. 21-26.

[BELA66] Belady, L. A., "A Study of Replacement Algorithms for Virtual Storage Computers," IBM Systems J., Vol. 5, No. 2, 1966, pp. 78-101.

[BELA69] Belady, L. A. and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," CACM, Vol. 12, No. 5, May 1969, pp. 282-288.

[BOBR67] Bobrow, D. G. and D. L. Murphy, "Structure of a LISP System Using Two-Level Storage," CACM, Vol. 10, No. 3, March 1967, p. 155.

[BRAW68] Brawn, B. and F. Gustavson, "Program Behavior in a Paging Environment," AFIPS FJCC, Vol. 33, 1968, pp. 1019-1032.

[BRAW70] Brawn, B. and F. Gustavson, "Sorting in a Paging Environment," CACM, Vol. 13, No. 8, August 1970, p. 483.

[BUDZ77] Budzinski, R. L., "Dynamic Memory Allocation for a Virtual Memory Computer," Coordinated Science Laboratory, University of Illinois at Champaign-Urbana, Report No. R-754, January 1977.

[CHU72] Chu, W. W. and H. Opderbeck, "The Page Fault Frequency Replacement Algorithm," AFIPS FJCC, Vol. 41, 1972, pp. 597-609.

[COME67] Comeau, L. W., "A Study of the Effect of User Program Optimization is a Paging System," Proc. ACM Symposium on Operating Systems Principles, Gatlinburg, Tenn., 1967.

[DENN68] Denning, P. J., "The Working Set Model for Program Behavior," CACM, Vol. 11, No. 5, May 1968, pp. 323-333.

[DENN70] Denning, P. J., "Virtual Memory," Computing Surveys, Vol. 2, No. 3, September 1970, pp. 153-189.

[DENN72a] Denning, P. J., "On Modeling Program Behavior," Proc. AFIPS SJCC, 1972, pp. 937-945.

[DENN72b] Denning, P. J. and J. R. Spirn, "Experiments with Program Localities," AFIPS FJCC, 1972, pp. 611-621.

[DENN75] Denning, P. J. and K. C. Kahn, "A Study of Program Locality and Lifetime Functions," Proc. 5th ACM Symposium on Operating System Principles, Austin, Texas, 1975, pp. 207-216.

[DUBR72] Dubrulle, A. A., "Solution of the Complete Symmetric Eigenvalue Problem in a Virtual Memory Environment," IBM JRD, November 1972, pp. 612-615.

[ELSH74] Elshoff, J. L., "Some Programming Techniques for Processing Multi-Dimensional Matrices in a Paging Environment," Proc. NCC, 1974, pp. 185-193.

[FERR74] Ferrari, D., "Improving Program Locality by Strategy-Oriented Restructuring," Information Processing 74 (Proc. IFIP Congress 74), North-Holland, Amsterdam, 1974, pp. 266-270.

[FERR75] Ferrari, D., "Tailoring Programs to Models of Program Behavior," IBM JRD, Vol. 19, No. 3, May 1975, pp. 244-251.

[FERR76a] Ferrari, D. and E. Lau, "An Experiment in Program Restructuring for Performance Enhancement," Proc. 2nd Int. Conf. on Software Engineering, San Francisco, Calif, October 1976.

[FERR76b] Ferrari, D., "The Improvement of Program Behavior," Computer, November 1976, pp. 39-47.

[FINE66] Fine, G. H., P. V. McIsaac, and C. W. Jackson, "Dynamic Program Behavior under Paging," Proc. ACM 21st Nat. Conf. 1966, Thompson Book Co., Washington, D. C., pp. 223-228.

[FRAN78] Franklin, M. A., G. S. Graham, and R. K. Gupta," Anomalies with Variable Partition Paging Algorithms," CACM, Vol. 21, No. 3, March 1978, pp. 232-236.

[FULL78] Fuller, S. H. and P. F. McGehearty, "Minimizing Latency in CCD Memories," IEEETC, Vol. C-27, No. 3, March 1978, pp. 252-254.

[GLAS65] Glaser, E. L. and J. B. Dennis, "The Structure of On-Line Information Processing Systems," Proc. Second Congress on Information Systems Sciences, 1965, pp. 5-14.

[HATF71] Hatfield, D. J. and J. Gerald, "Program Restructuring for Virtual Memory," IBM Systems Journal, Vol. 10, No. 3, 1971, pp. 168-192.

[IBM73] "Introduction to Virtual Storage in System/370," IBM Publication GR20-4260-1, February 1973, pp. 50-51.

[JONE72] Jones, P. D., "Implicit Storage Management in the Control Data STAR-100," COMPCON 72 Digest, 1972, pp. 5-7.

[JULI78] Juliussen, J.E., "Bubbles and CCD Memories-Solid State Mass Storage," Proc. NCC, 1978, pp. 1067-1075.

[KILB62] Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-Level Storage System," IRE Transactions, EC-11-Vol. 2, April 1962, pp. 223-235.

[KUCK70] Kuck, D. J. and D. H. Lawrie, "The Use and Performance of Memory Hierarchies: A Survey," Software Engineering, Vol. 1, 1970, pp. 45-77.

[KUCK78] Kuck, D. J., "The Structure of Computers and Computation," John Wiley and Son Inc., 1978.

[LEAS76] Leasure, B. R., "Compiling Serial Languages for Parallel Machines," Department of Computer Science, University of Illinois at Champaign-Urbana, Report No. 805, November 1976.

[MASU74] Masuda, T., H. Shiota, K. Noguchi, and T. Ohki, "Optimization of Program Organization by Cluster Analysis," Information Processing 74 (Proc. IFIP Congress 74), North-Holland, Amsterdam, 1974, pp. 261-265.

[McKE69] McKeller, A. C. and E. G. Coffman, "The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment," CACM, Vol. 12, No. 3, 1969, pp. 153-165.

[MOLE72] Moler, C. B., "Matrix Computation with Fortran and Paging," CACM, Vol. 15, No. 4, 1972, p. 268.

[PRIE76] Prieve, B. G. and R. S. Fabry, "VMIN-An Optimal Variable Space Replacement Algorithm," CACM, Vol. 19, No. 5, May 1976, pp. 295-297.

[ROGE73] Rogers, L. D., "Optimal Paging Strategies and Stability Considerations for Solving Large Linear Systems," Ph.D. Thesis, University of Waterloo, Canada, 1973.

[SAYR69] Sayre, D., "Is Automatic Folding of Programs Efficient Enough to Displace Manual?" CACM, Vol. 12, December 1969, pp. 656-660.

[SCHE73] Scherr, A. L., "The Design of IBM OS/VS2 Release 2," AFIPS Conf. Proc., Vol. 42, 1973, pp. 387-394.

[SHED72] Shedler, G. S. and C. Tung, "Locality in Page Reference Strings," SIAM J. Computing, Vol. 1, No. 3, September 1972, pp. 218-241.

[SITE78] Sites, R. L., "Optimal Shift Strategy for a Block-Transfer CCD Memory," CACM, Vol. 21, No. 3, May 1978, pp. 423-425.

[SMIT67] Smith, J. L., "Multiprogramming under a Page on Demand Strategy," CACM, Vol. 10, No. 10, October 1967, pp. 636-646.

[SMIT76] Smith, A. J., "A Modified Working Set Paging Algorithm," IEEETC, Vol. C-25, No. 9, September 1976, pp. 907-914.

[SPIR76] Spirn, J., "Distance String Models for Program Behavior," Computer, November 1976, pp. 14-20.

[SPIR77] Spirn, J., "Program Behavior: Models and Measurements," Elsevier-North Holland, N. Y., 1977.

[TOWL76] Towle, R. A., "Control and Data Dependence for Program Transformations," Department of Computer Science, University of Illinois at Champaign-Urbana, Report No. 788, March 1976.

[WOLF78] Wolfe, M. J., "Techniques for Improving the Inherent Parallelism in Programs," M. S. Thesis, Department of Computer Science, University of Illinois at Champaign-Urbana, February 1978.

APPENDIX

In this appendix, we show the page faults and the space-time cost curves for our untransformed and transformed programs. The replacement algorithm used is the LRU algorithm and the page size is 256 bytes. The space-time cost is measured in pages-page faults (see Section 4.2.2).

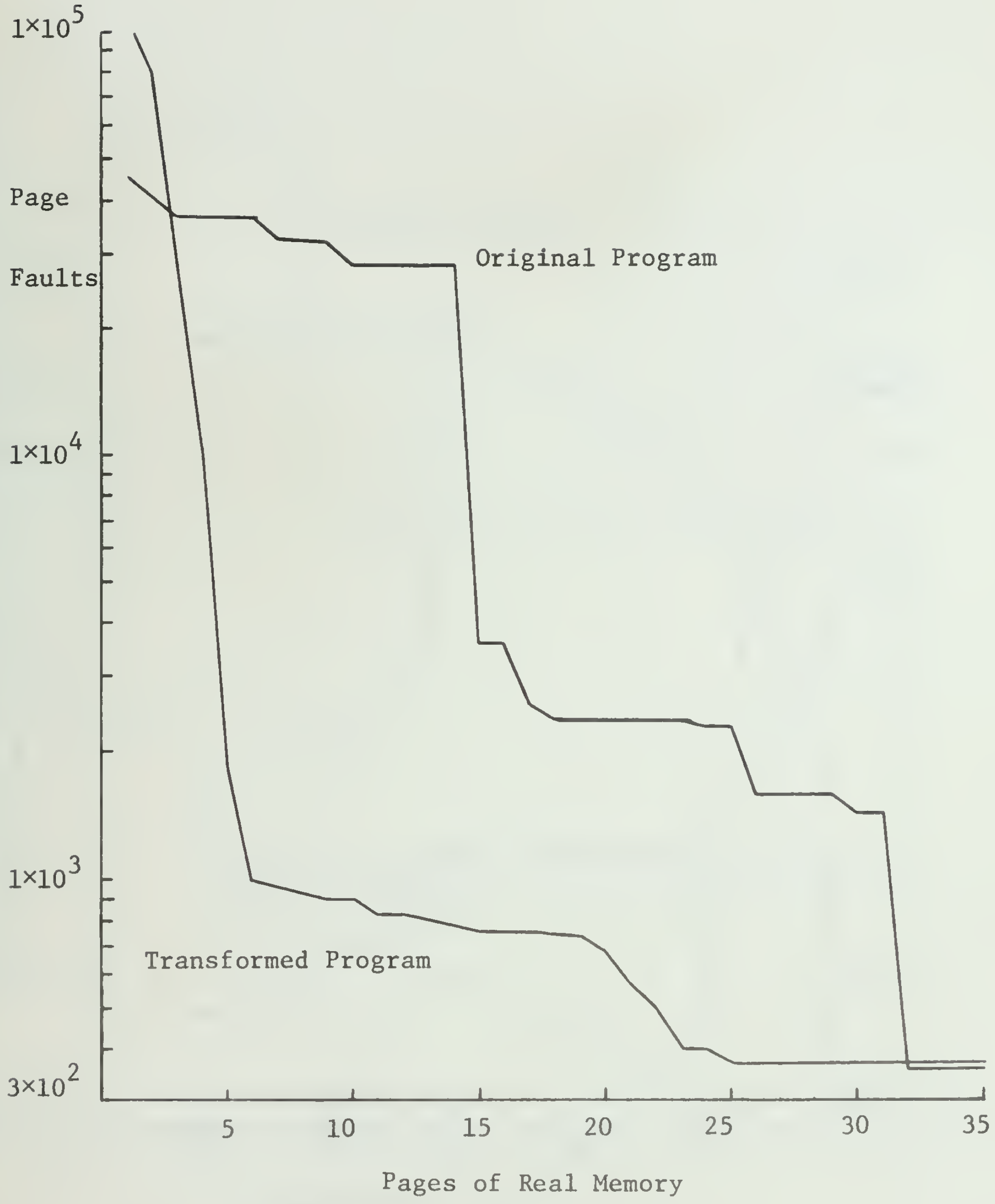


Figure 34-a. The Page Faults Curves for Program ADVECT

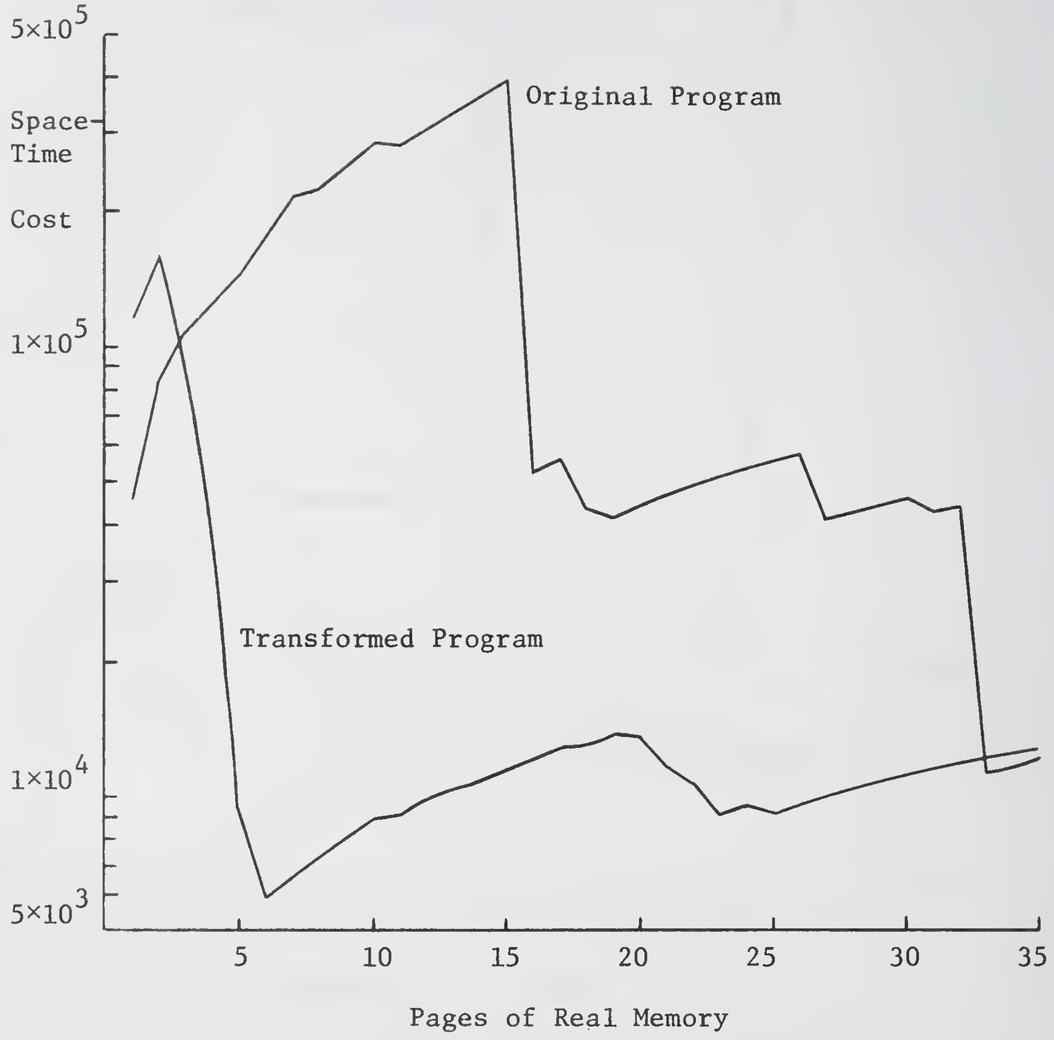


Figure 34-b. The Space-Time Cost Curves for Program ADVECT

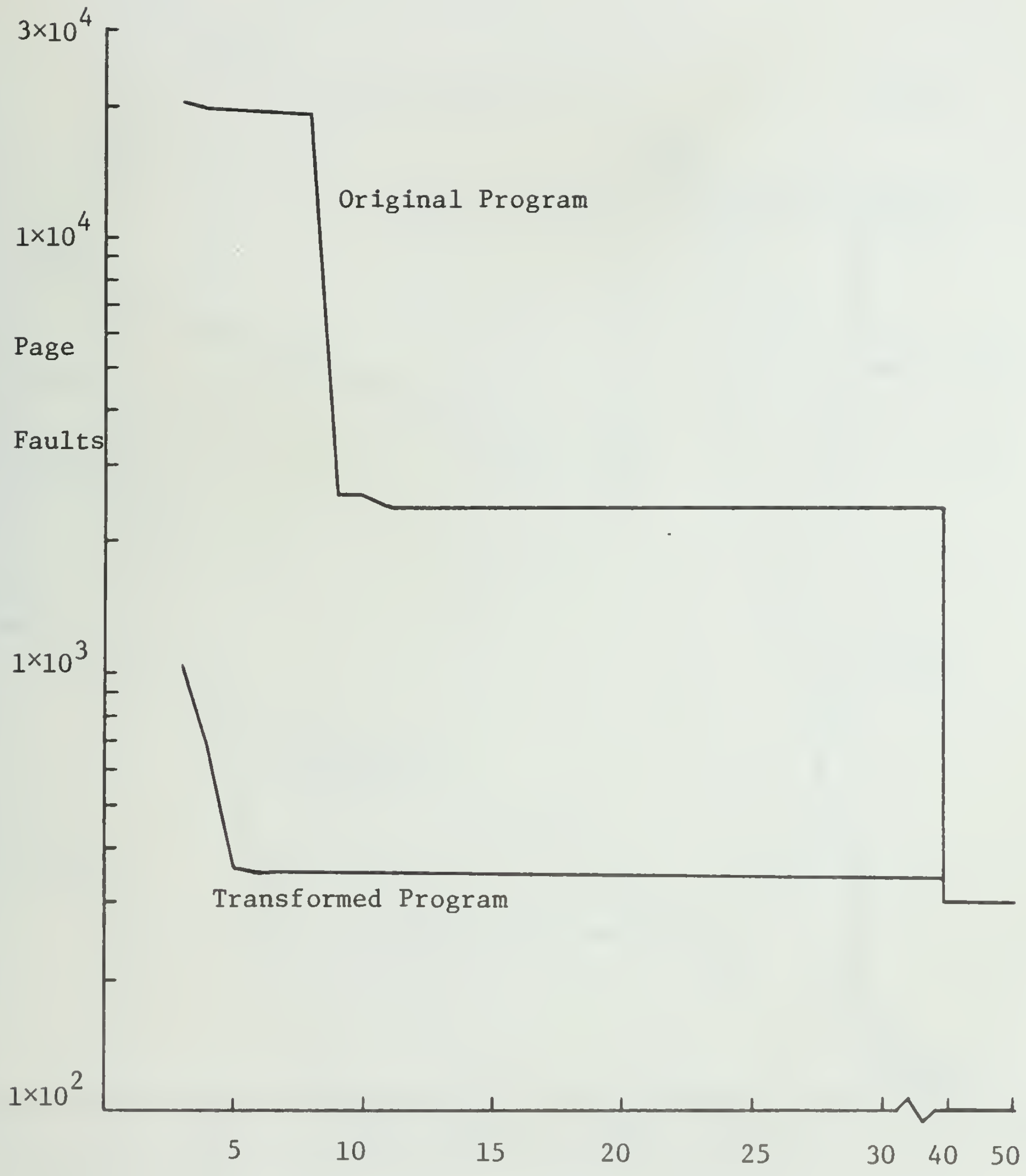


Figure 35-a. The Page Faults Curves for Program BASE

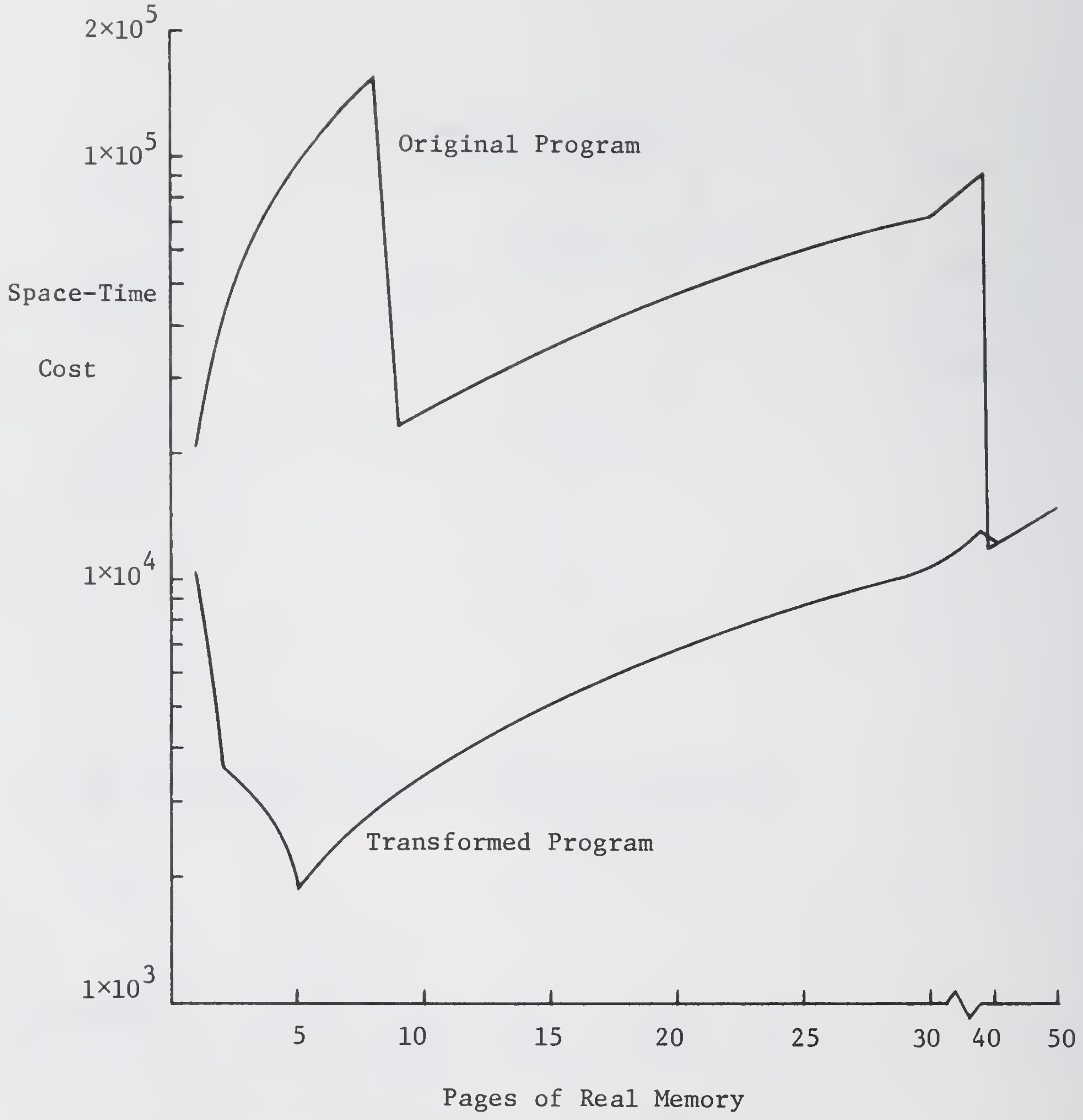


Figure 35-b. The Space-Time Cost Curves for Program BASE

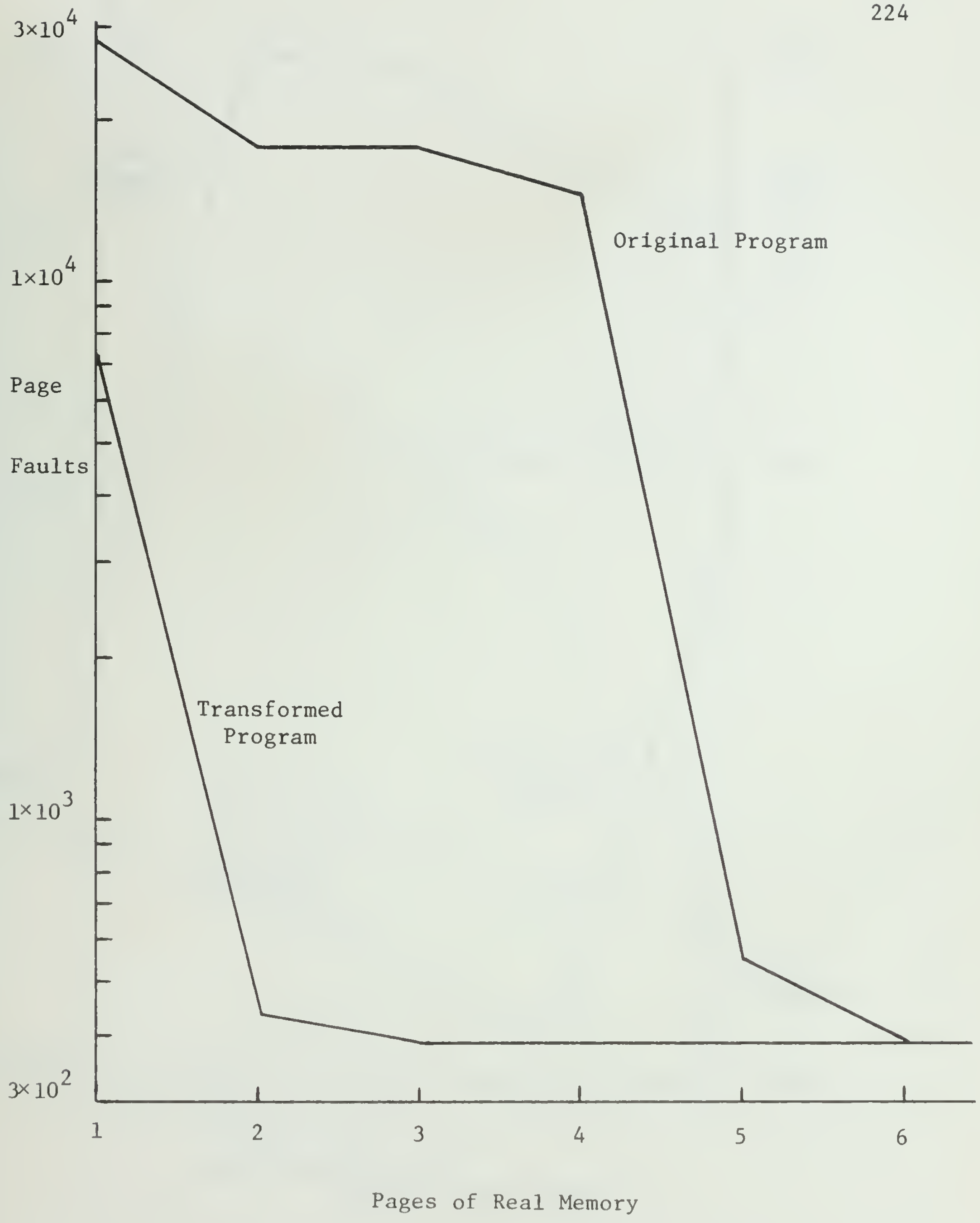


Figure 36-a. The Page Faults Curves for Program BIGEN

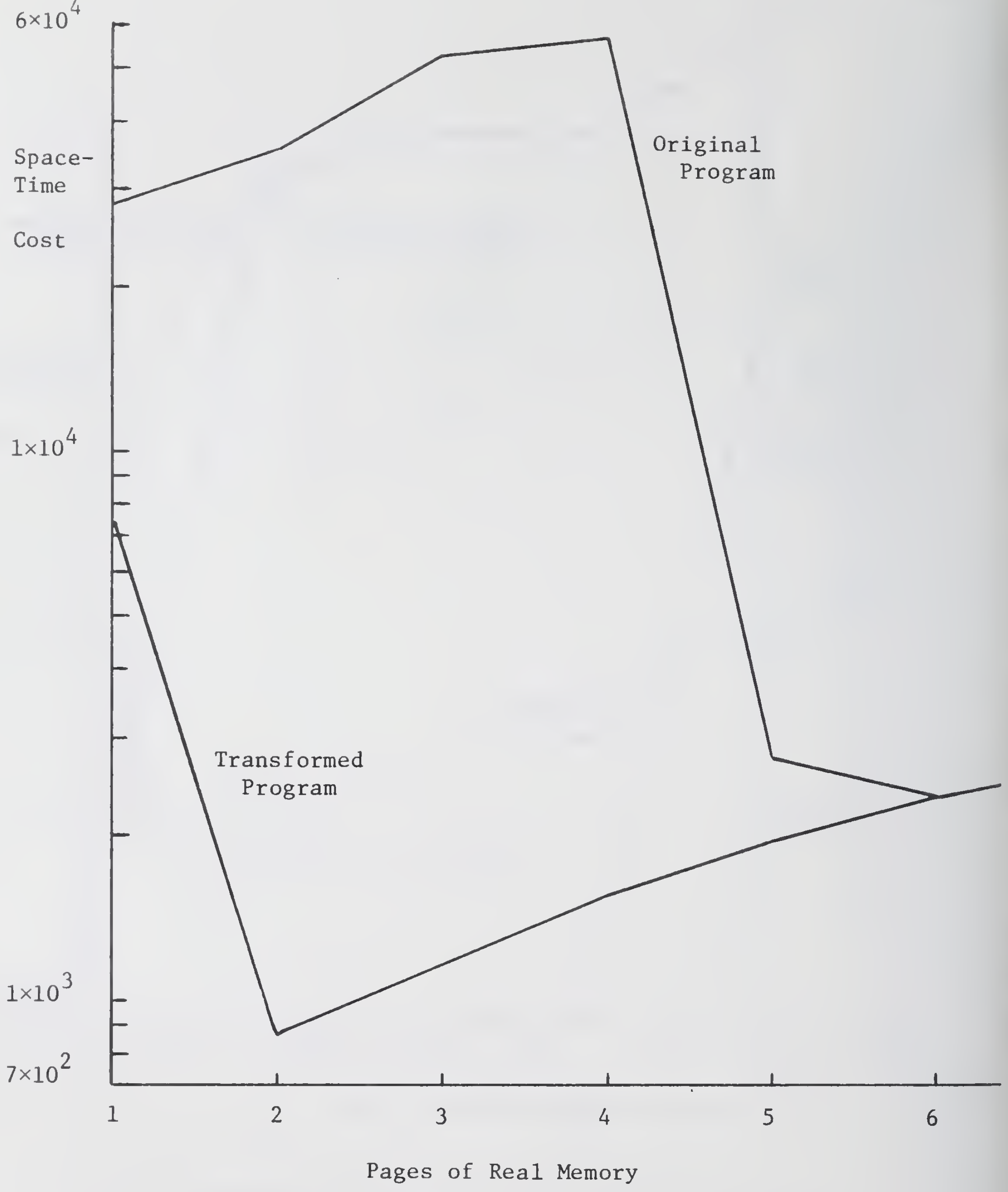


Figure 36-b. The Space-Time Cost Curves for Program BIGEN

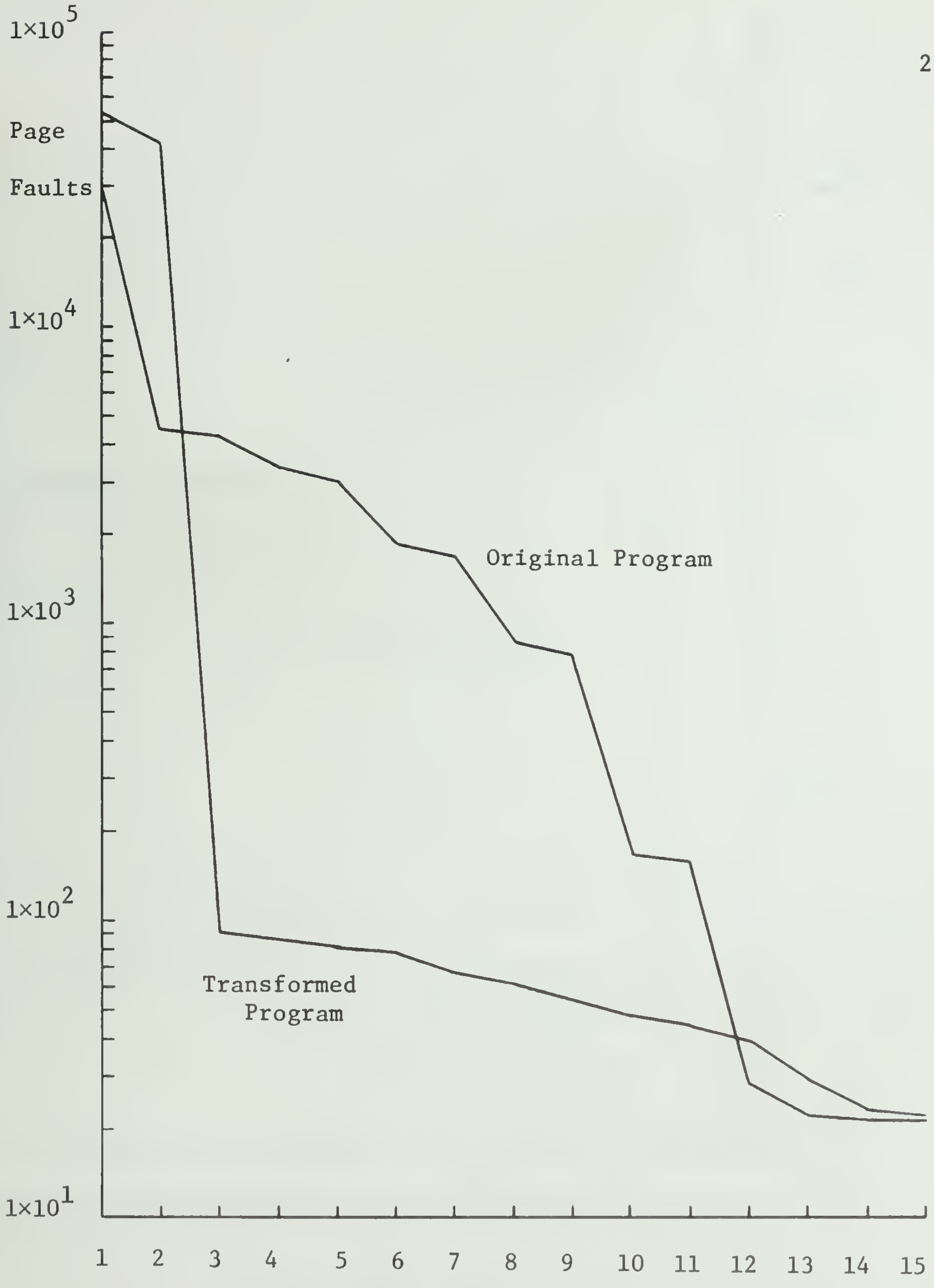


Figure 37-a. The Page Faults Curves for Program CD

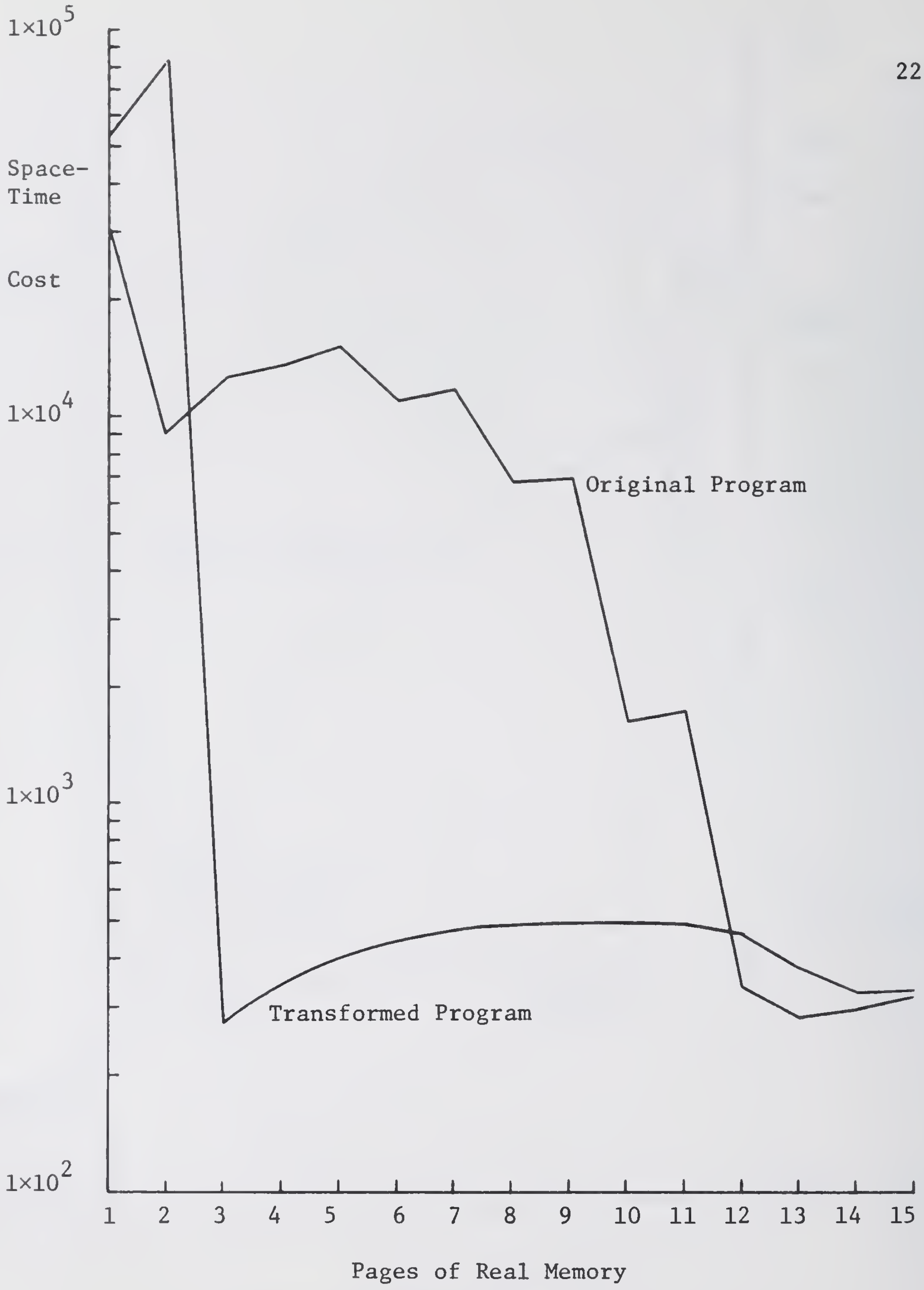


Figure 37-b. The Space-Time Cost Curves for Program CD

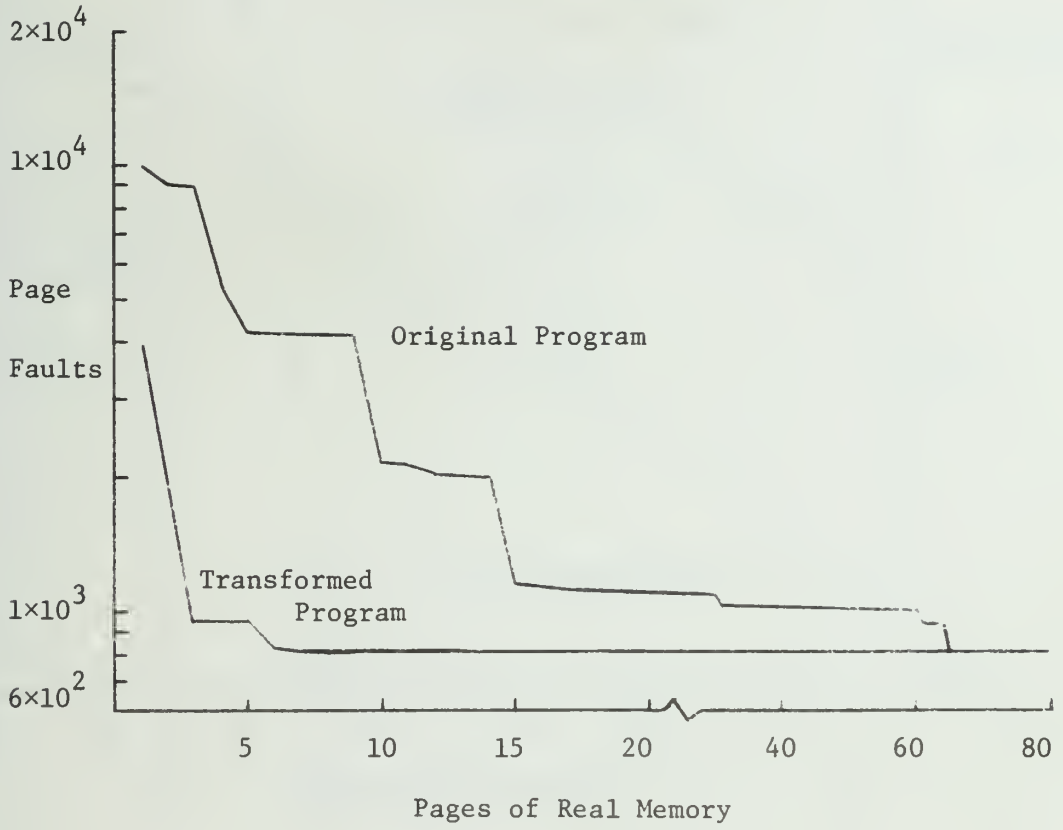


Figure 38-a. The Page Faults Curves for Program DISPERSE

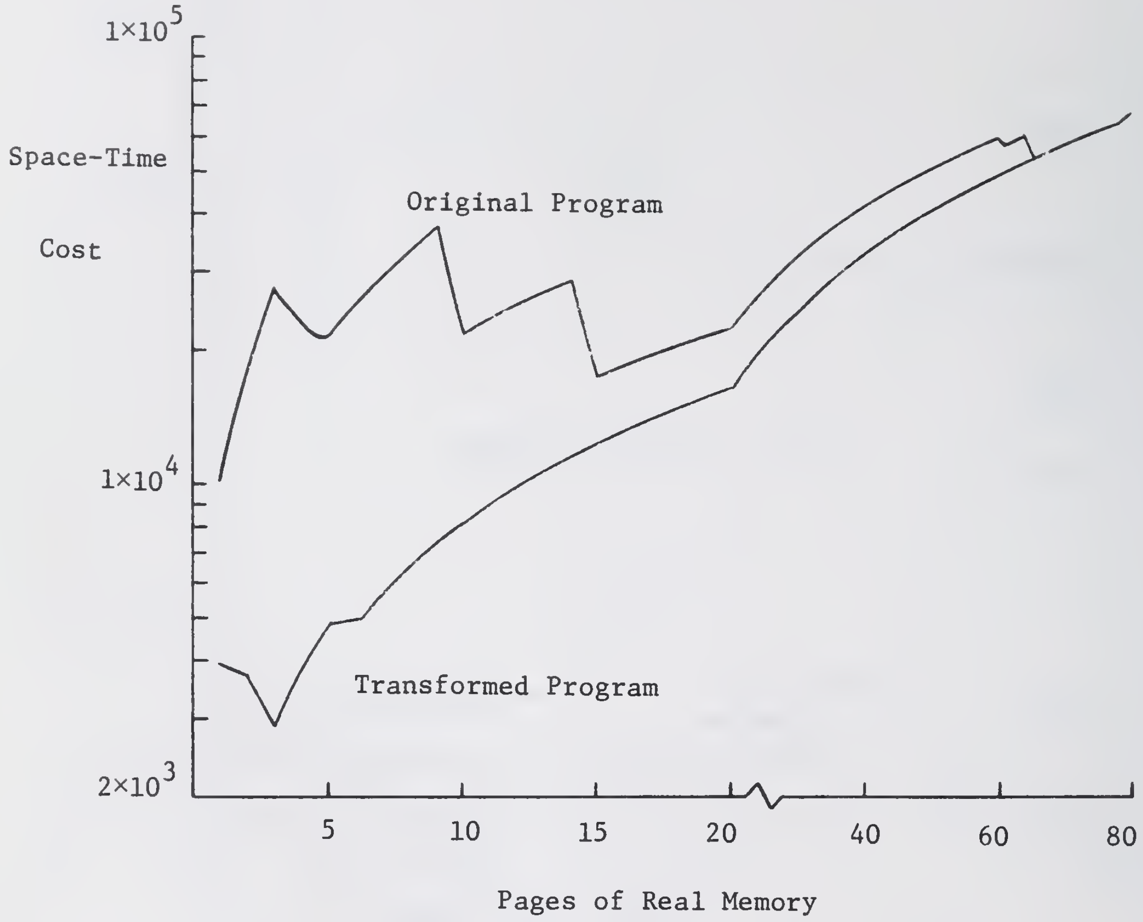


Figure 38-b. The Space-Time Cost Curves for Program DISPERSE

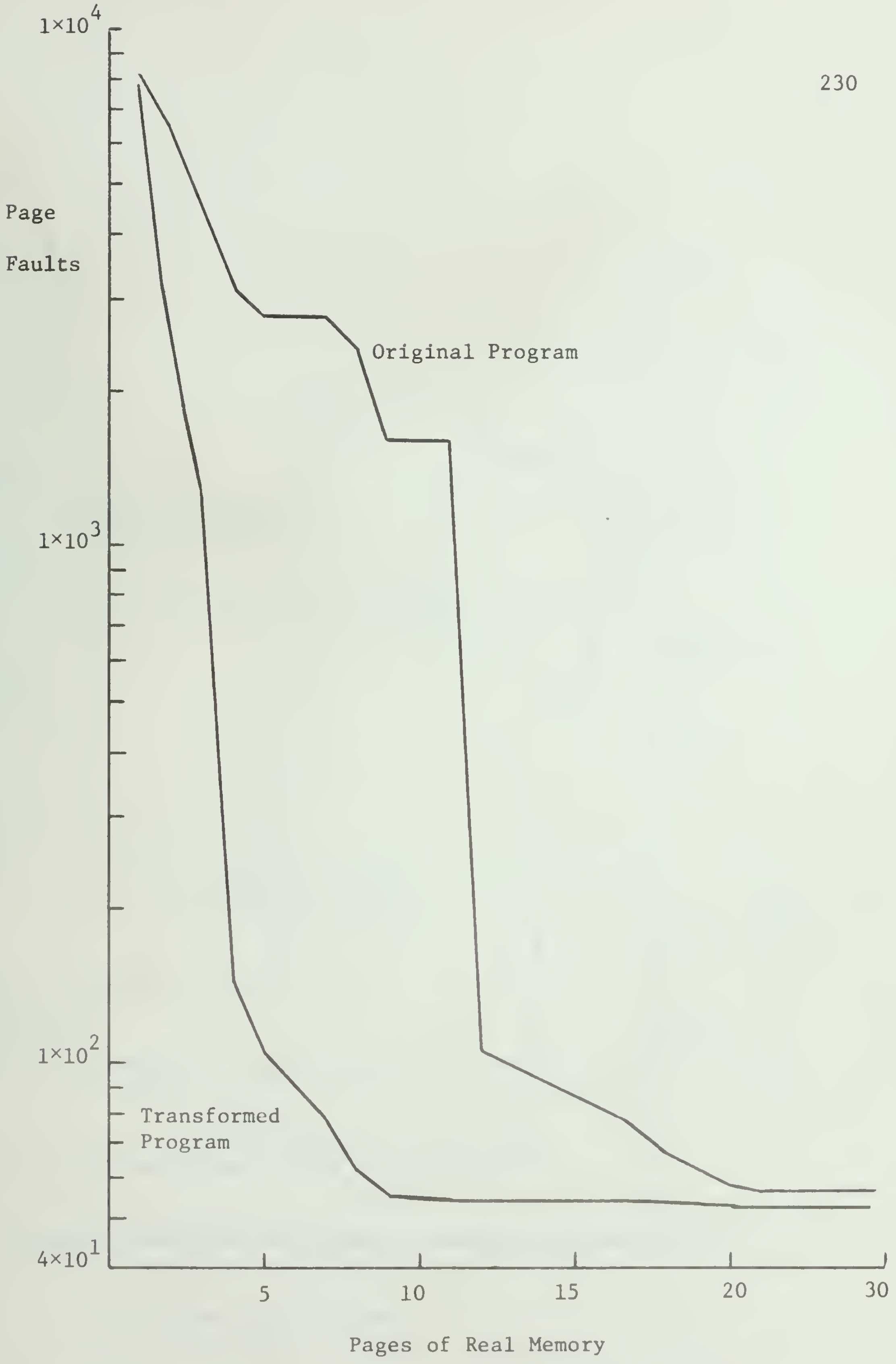


Figure 39-a. The Page Faults Curves for Program FIELD

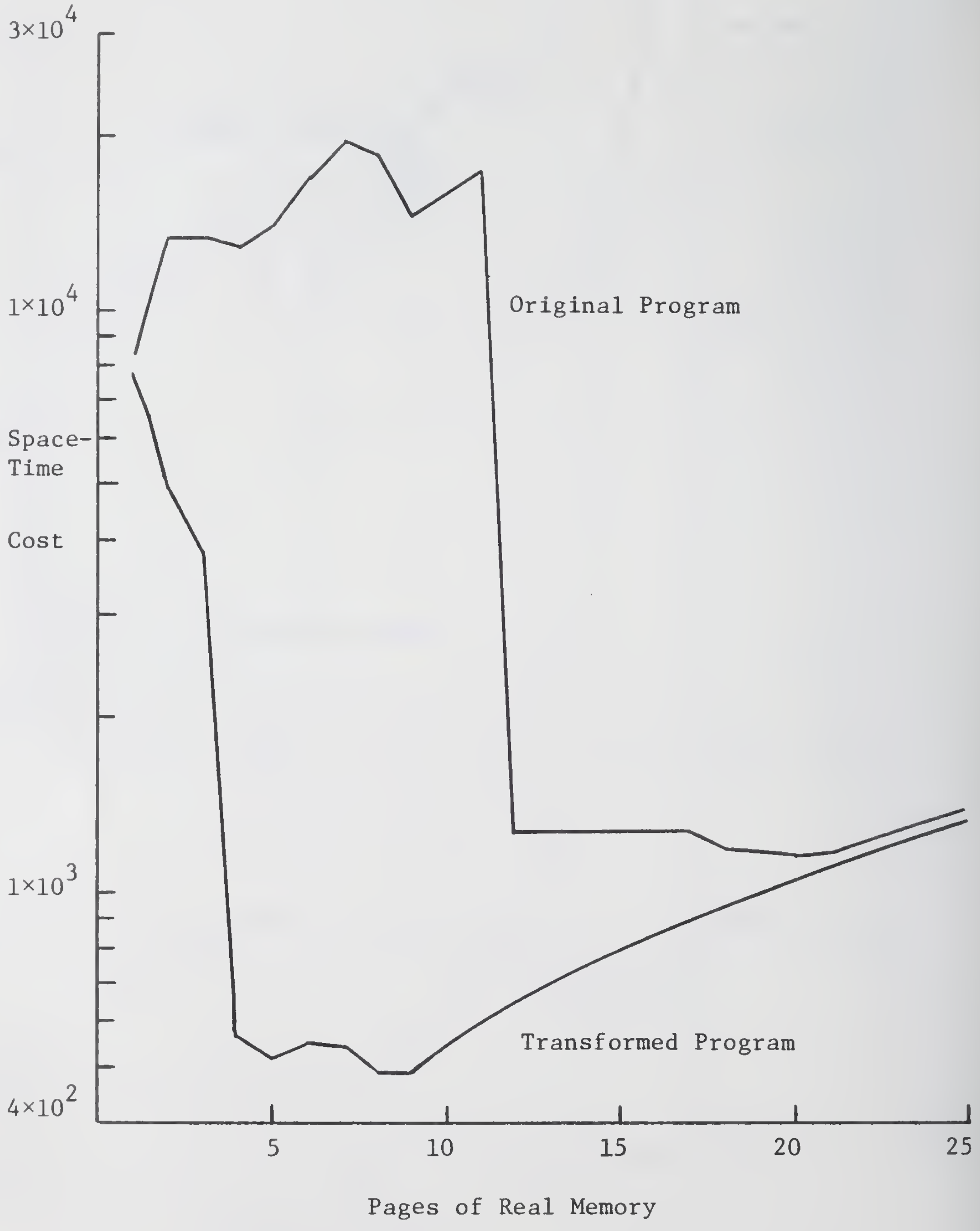


Figure 39-b. The Space-Time Cost Curves for Program FIELD

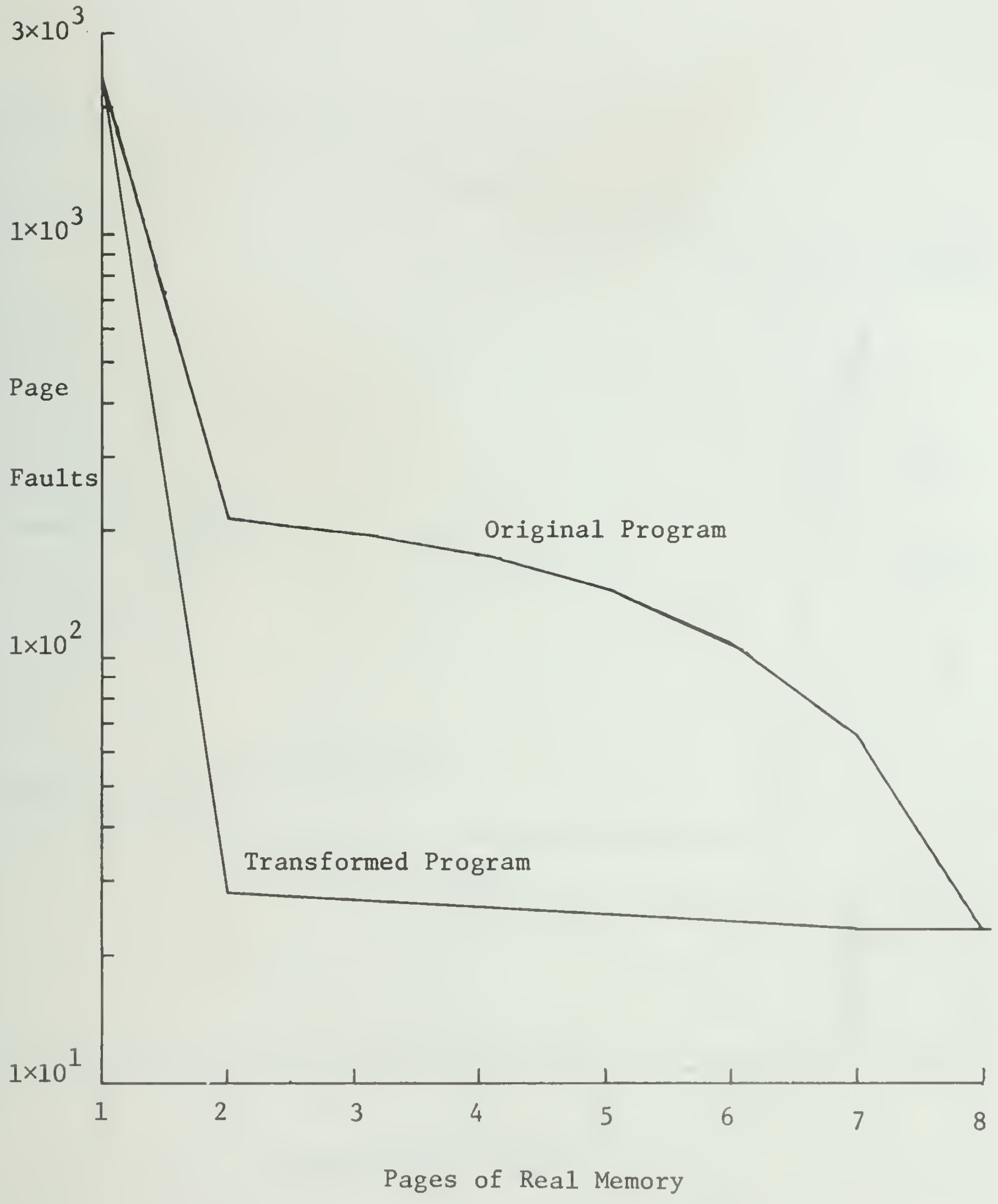


Figure 40-a. The Page Fault Curves for Program FLR

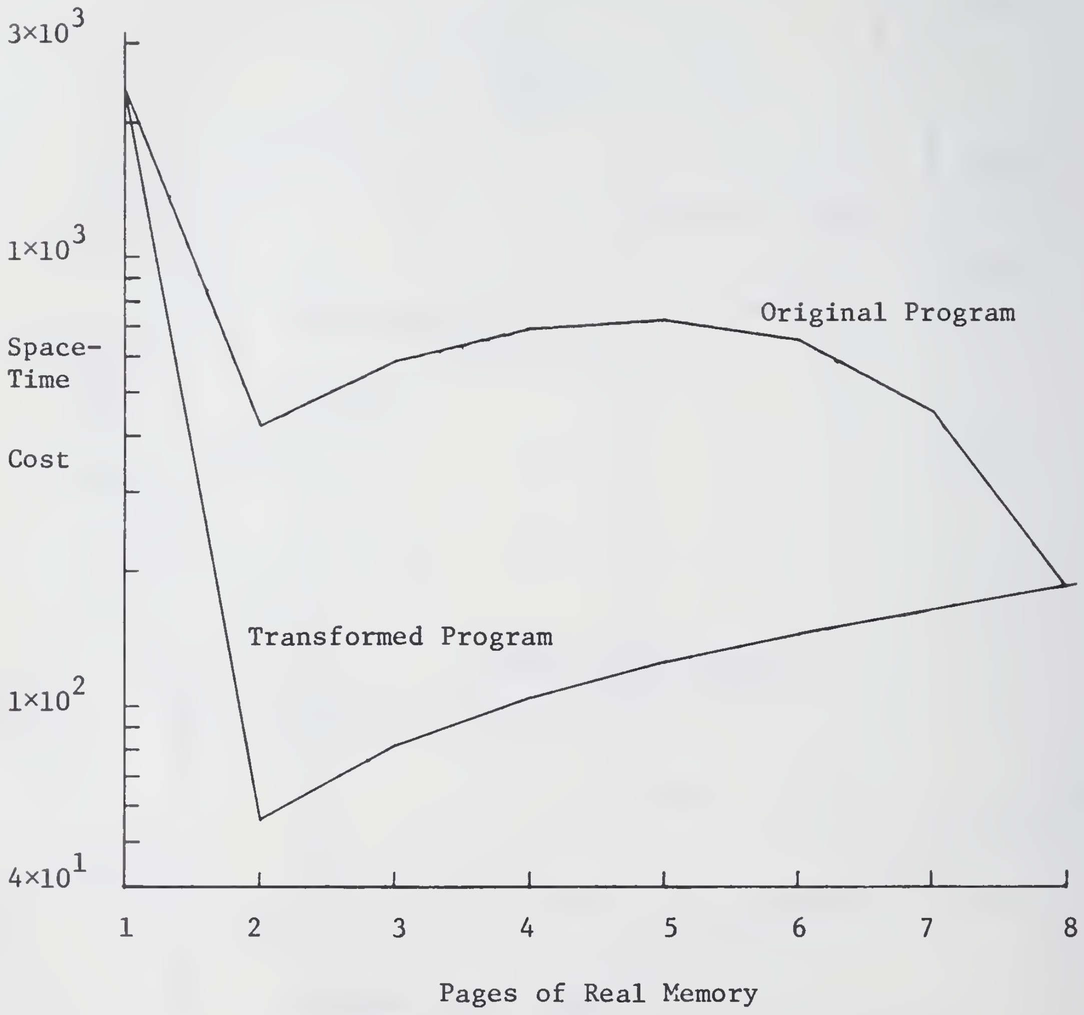


Figure 40-b. The Space-Time Cost Curves for Program FLR

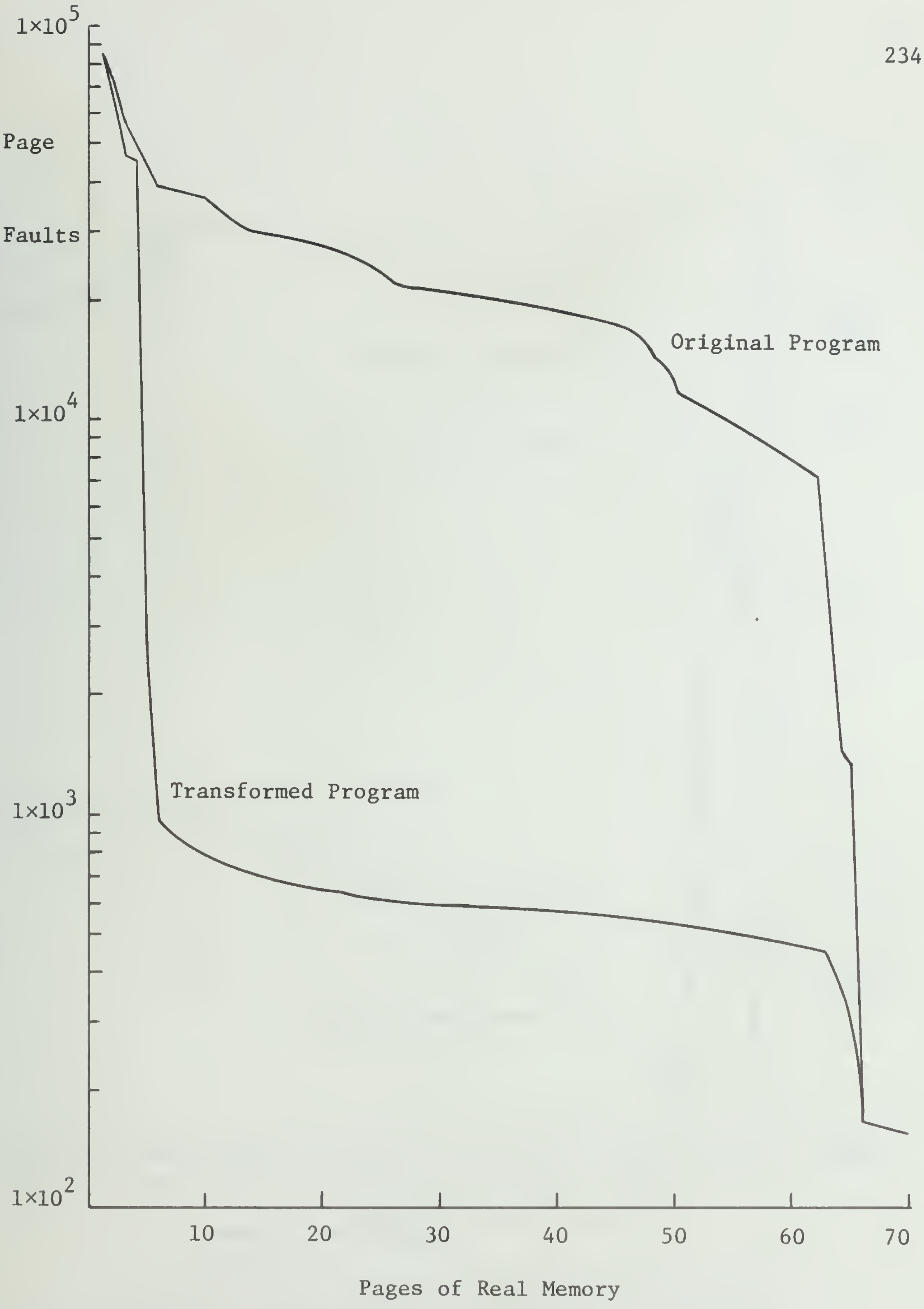


Figure 41-a. The Page Faults Curves for Program FOURTR

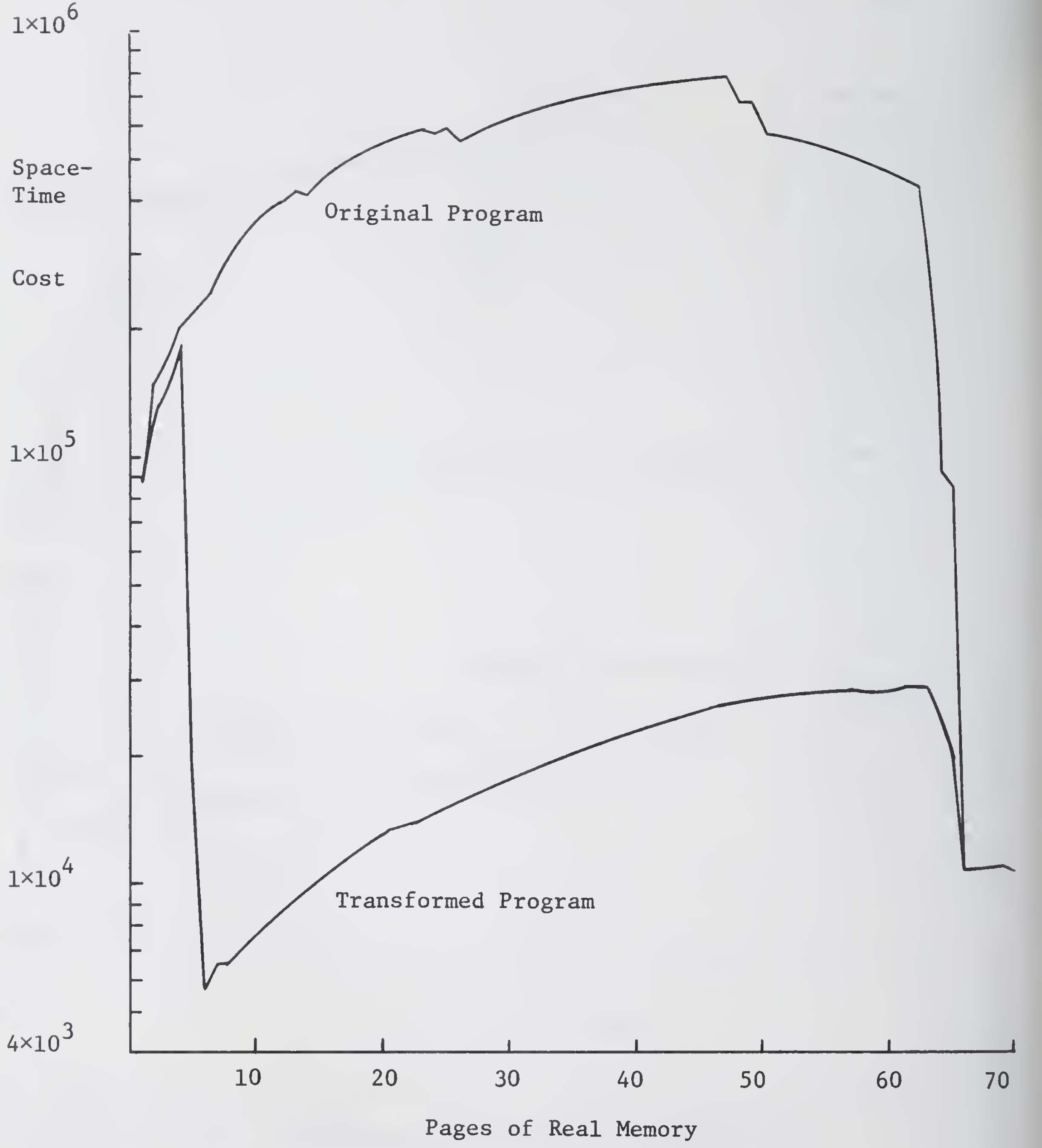


Figure 41-b. The Space-Time Cost Curves for Program FOURTR

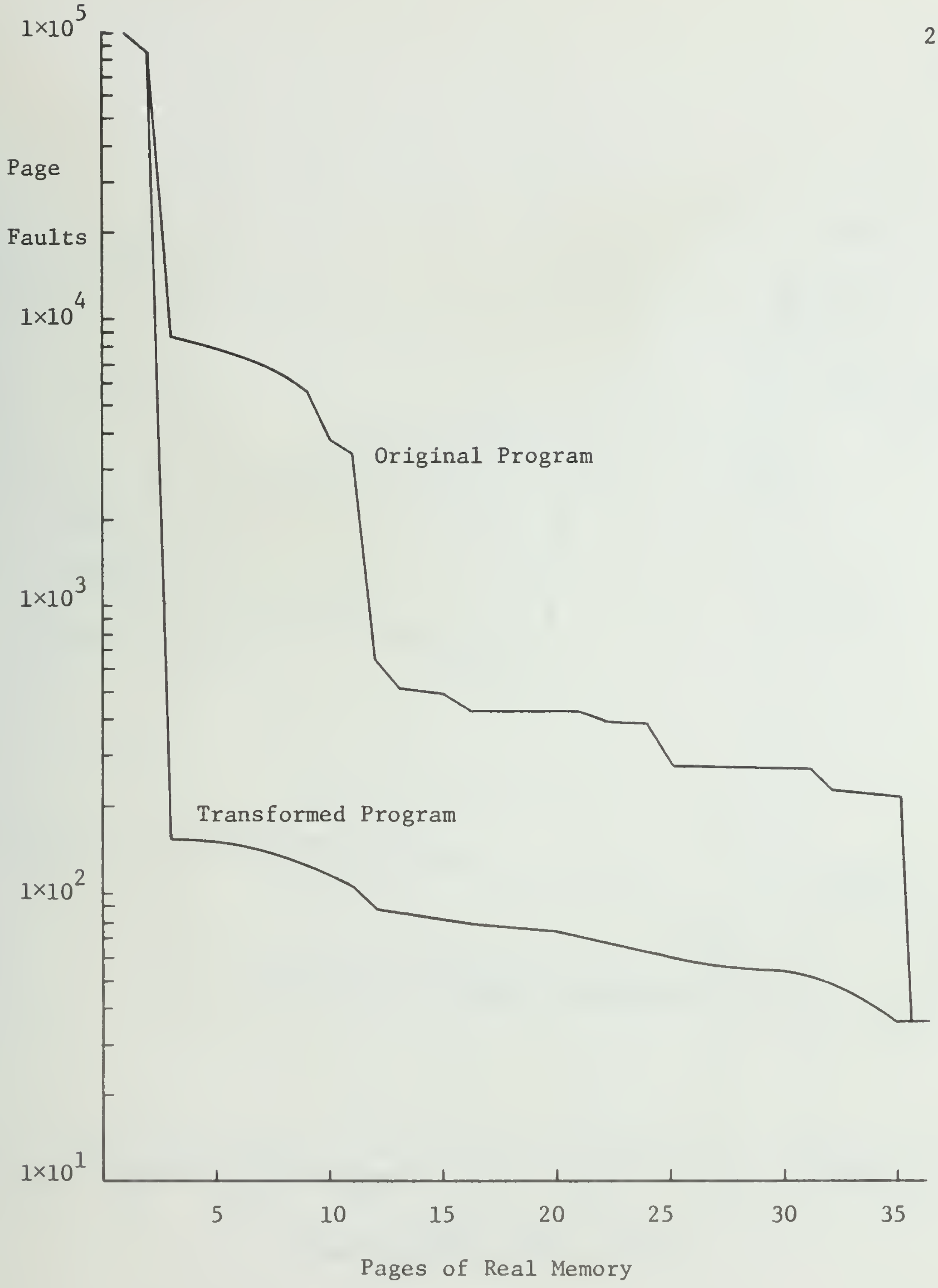


Figure 42-a. The Page Faults Curves for Program GE

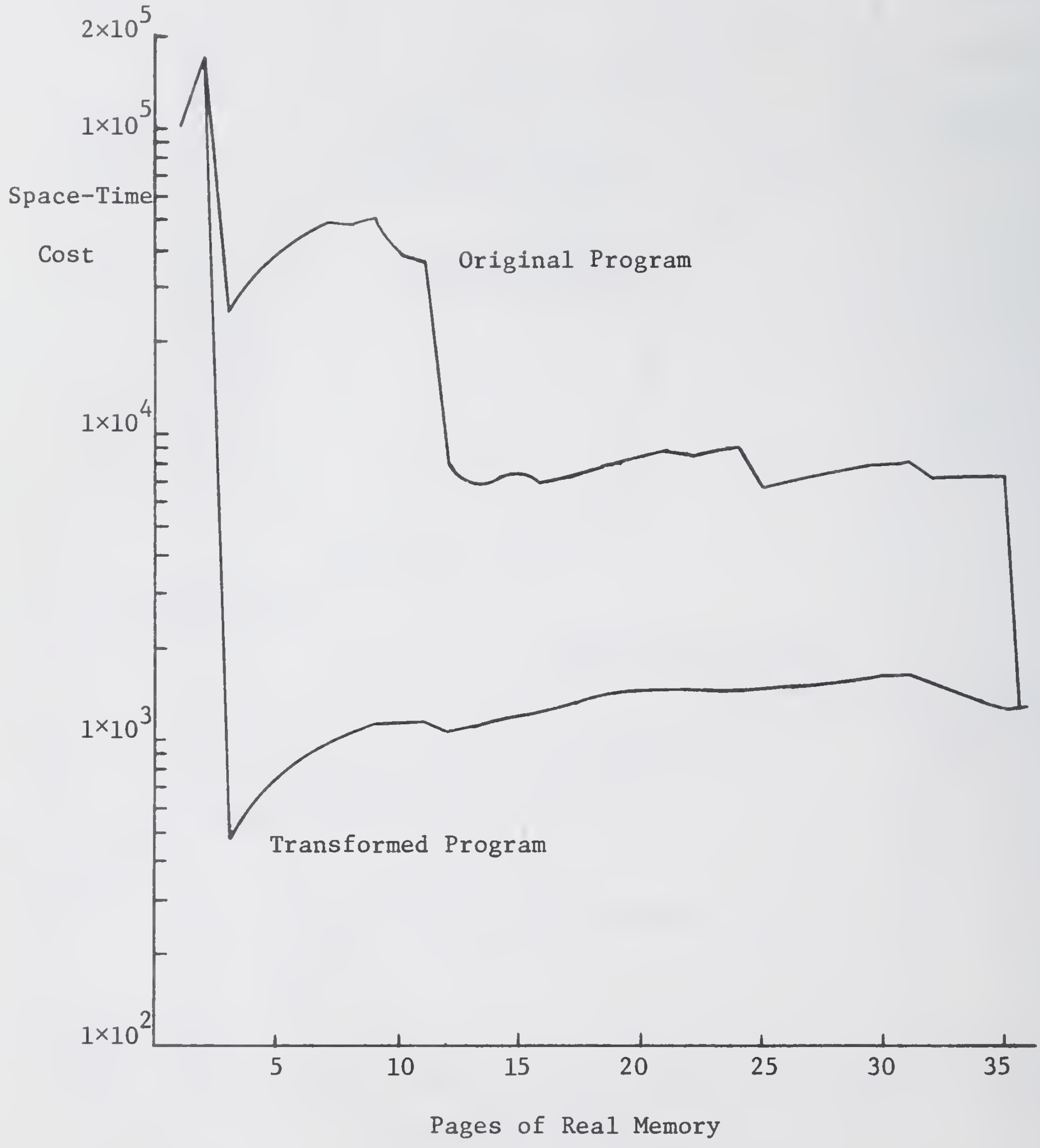


Figure 42-b. The Space-Time Cost Curves for Program GE

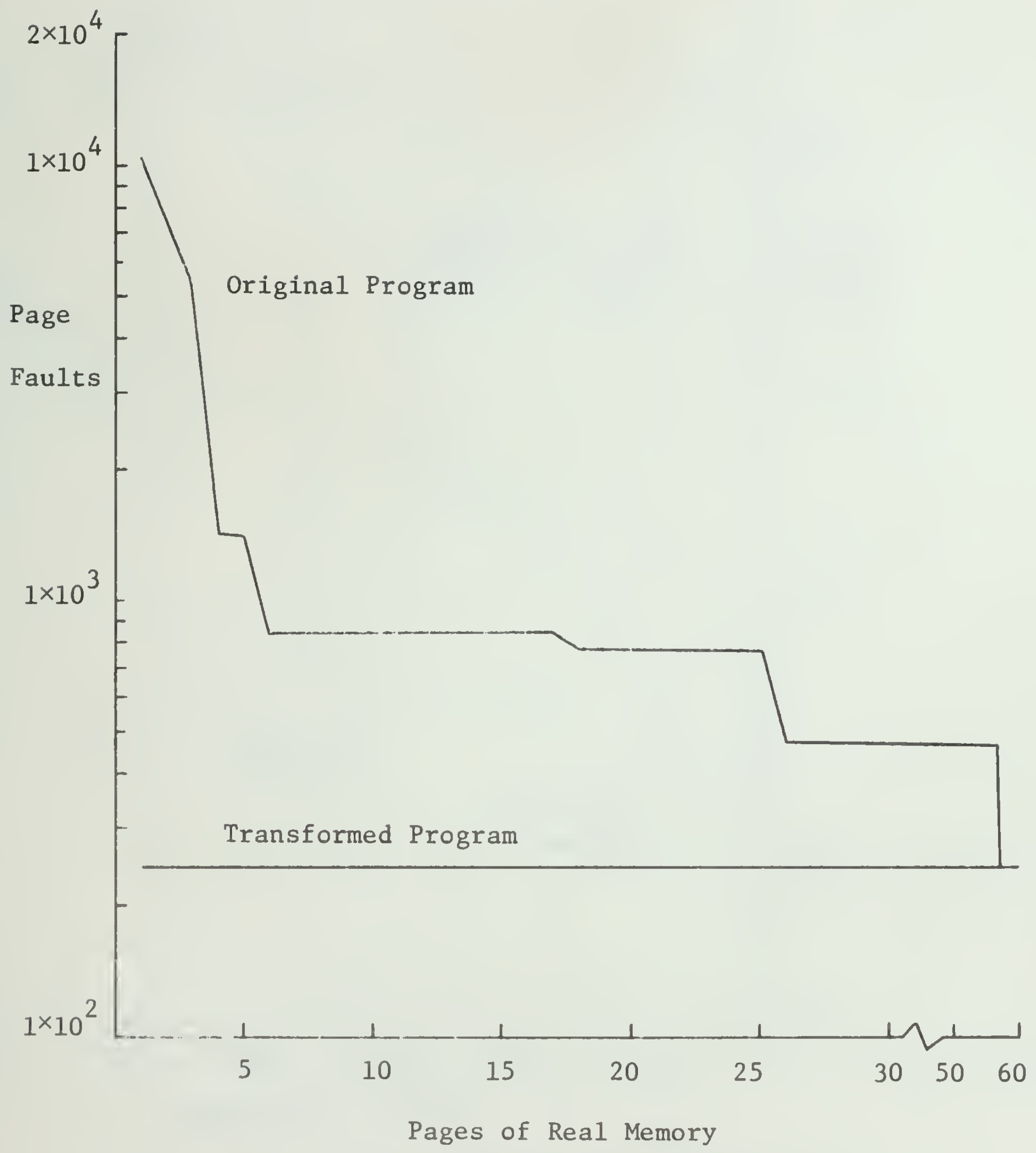


Figure 43-a. The Page Faults Curves for Program INIT

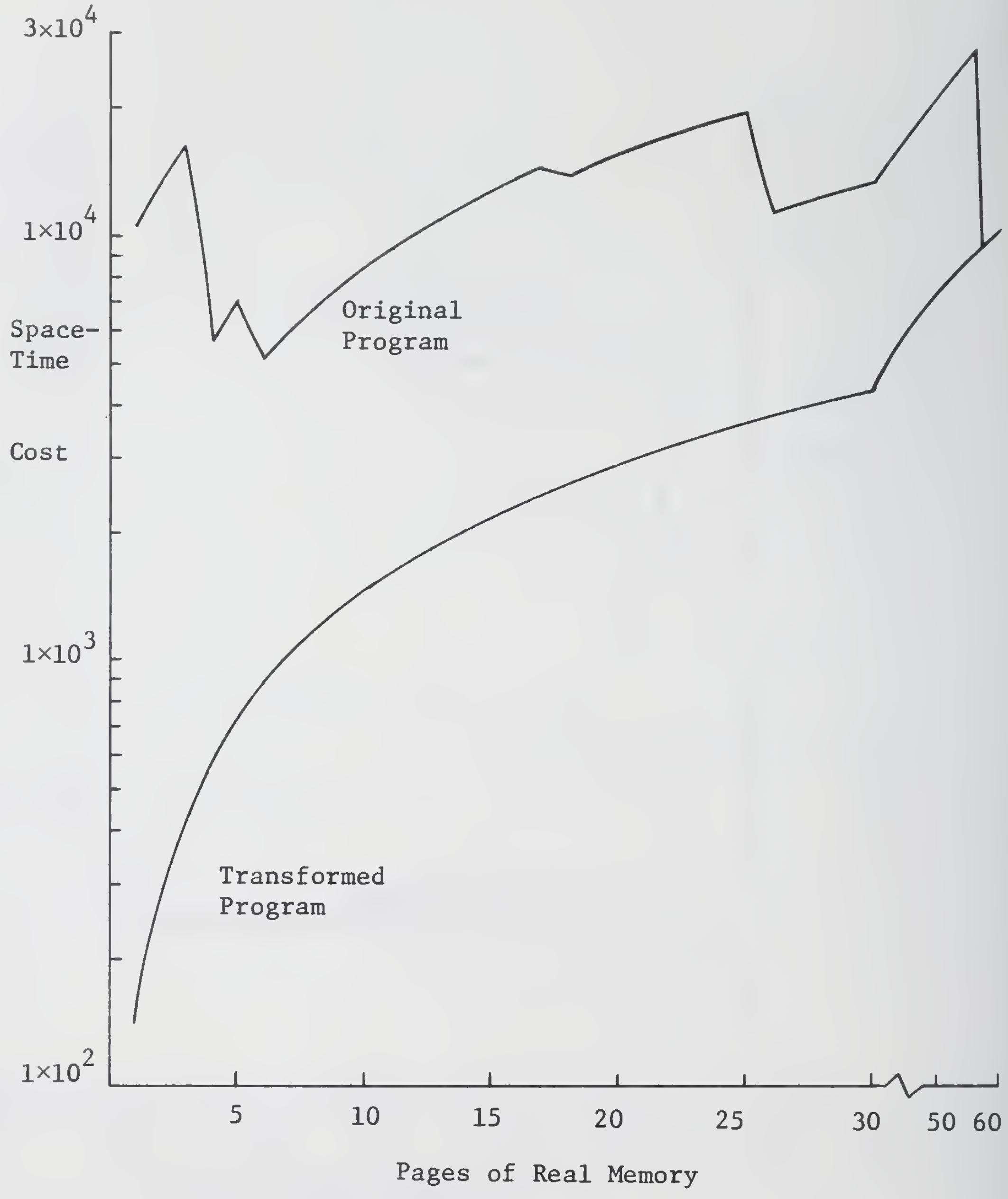


Figure 43-b. The Space-Time Cost Curves for Program INIT

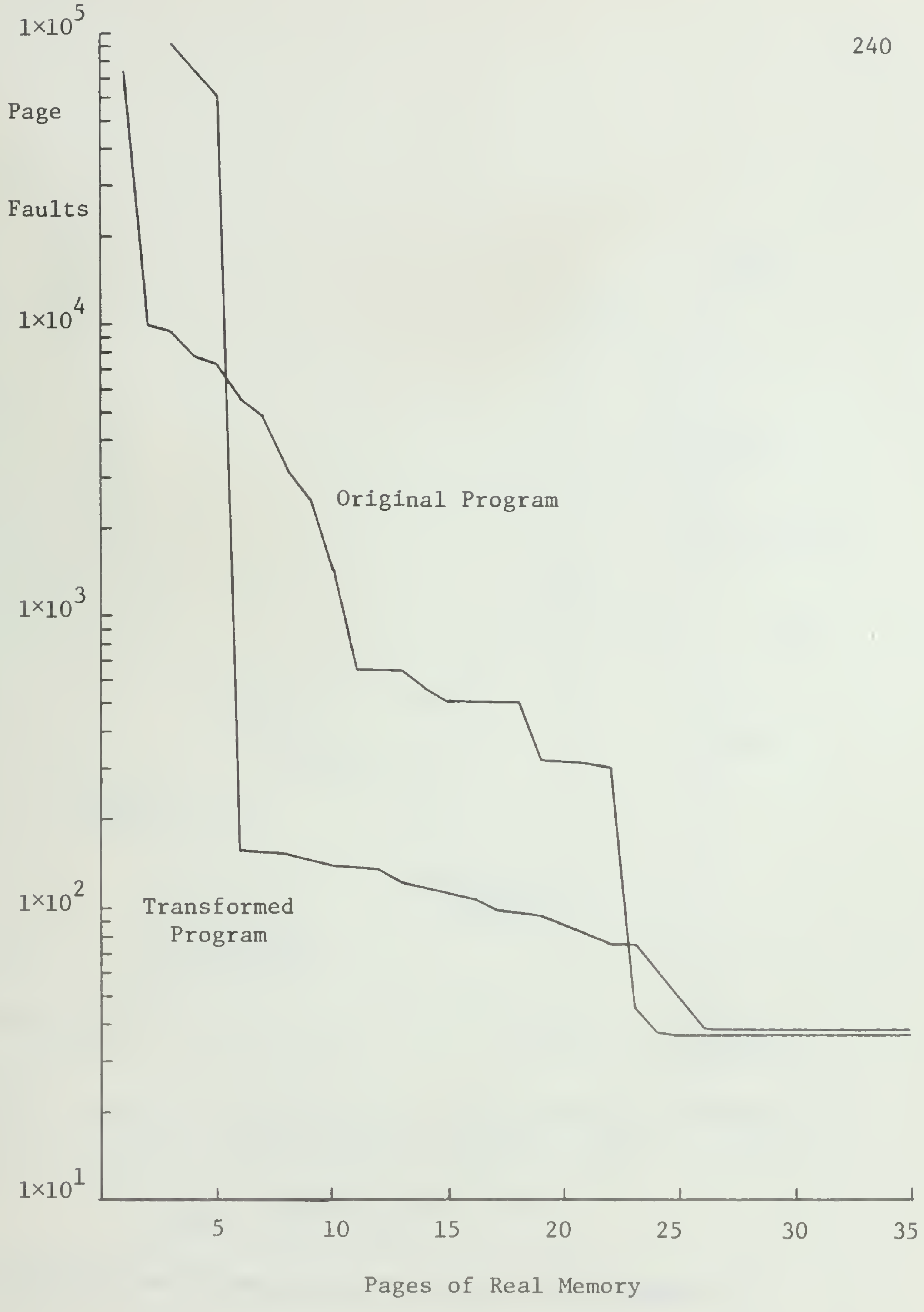


Figure 44-a. The Page Faults Curves for Program LUD

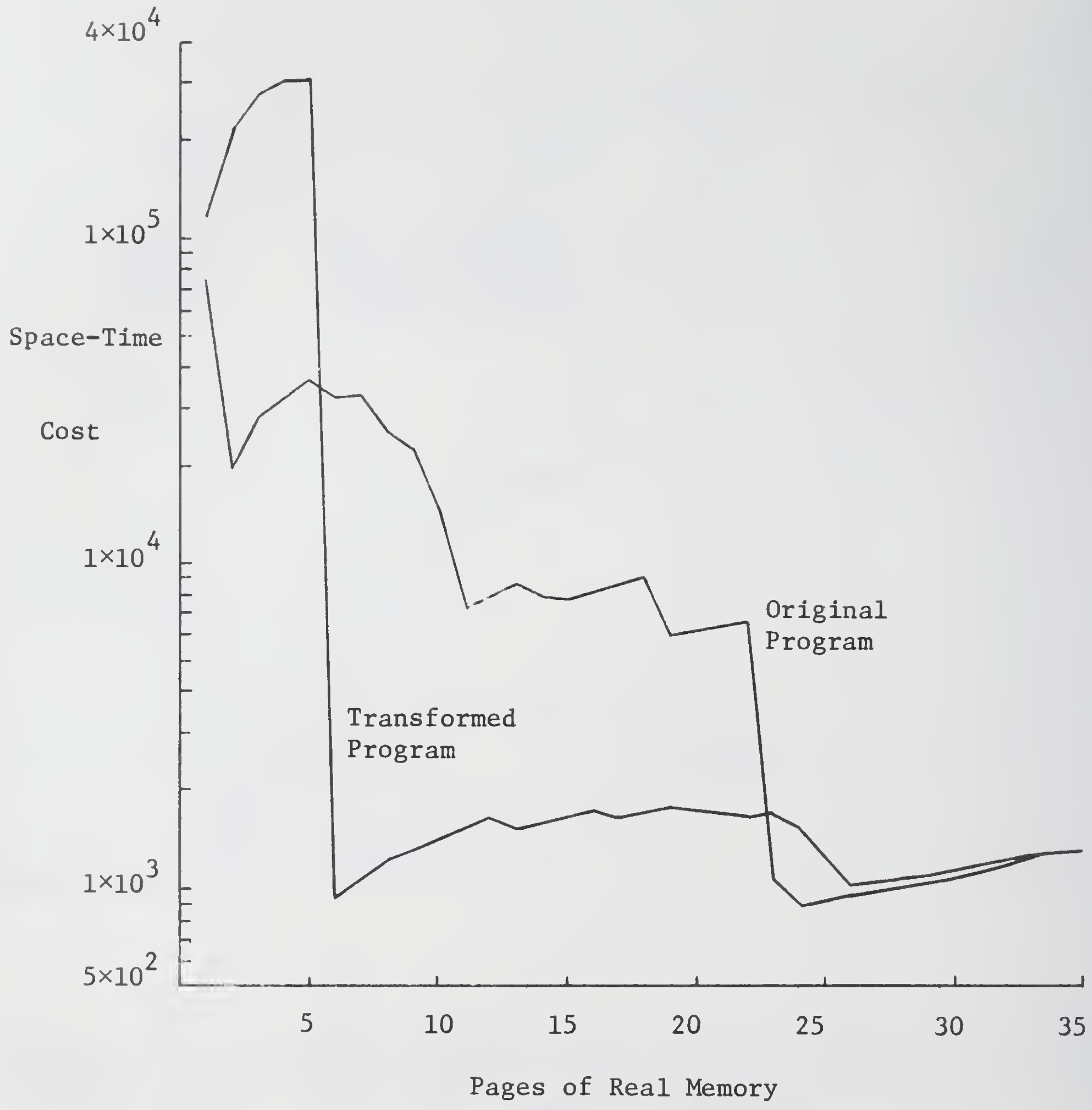


Figure 44-b. The Space-Time Cost Curves for Program LUD

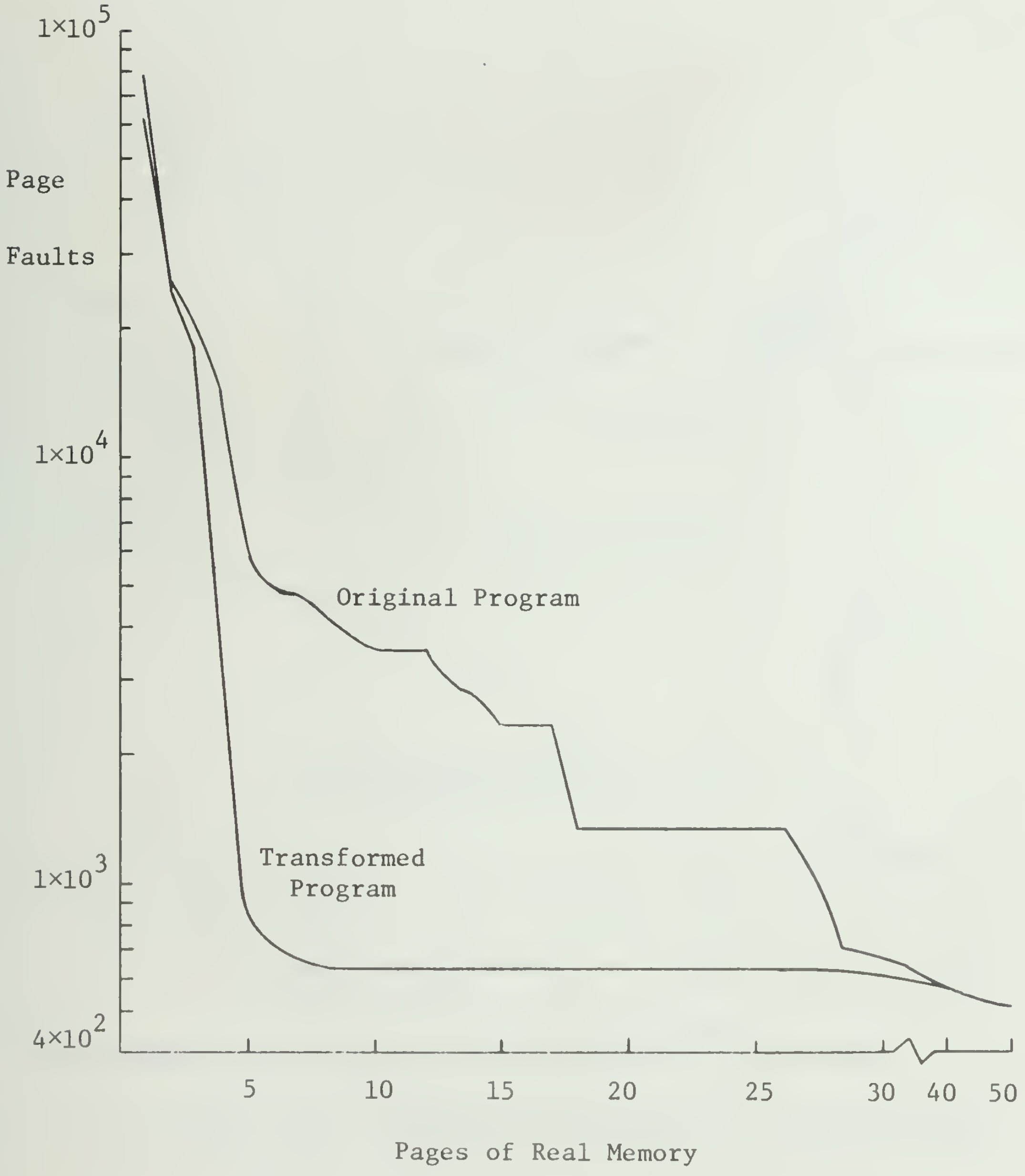


Figure 45-a. The Page Faults Curves for Program MAIN

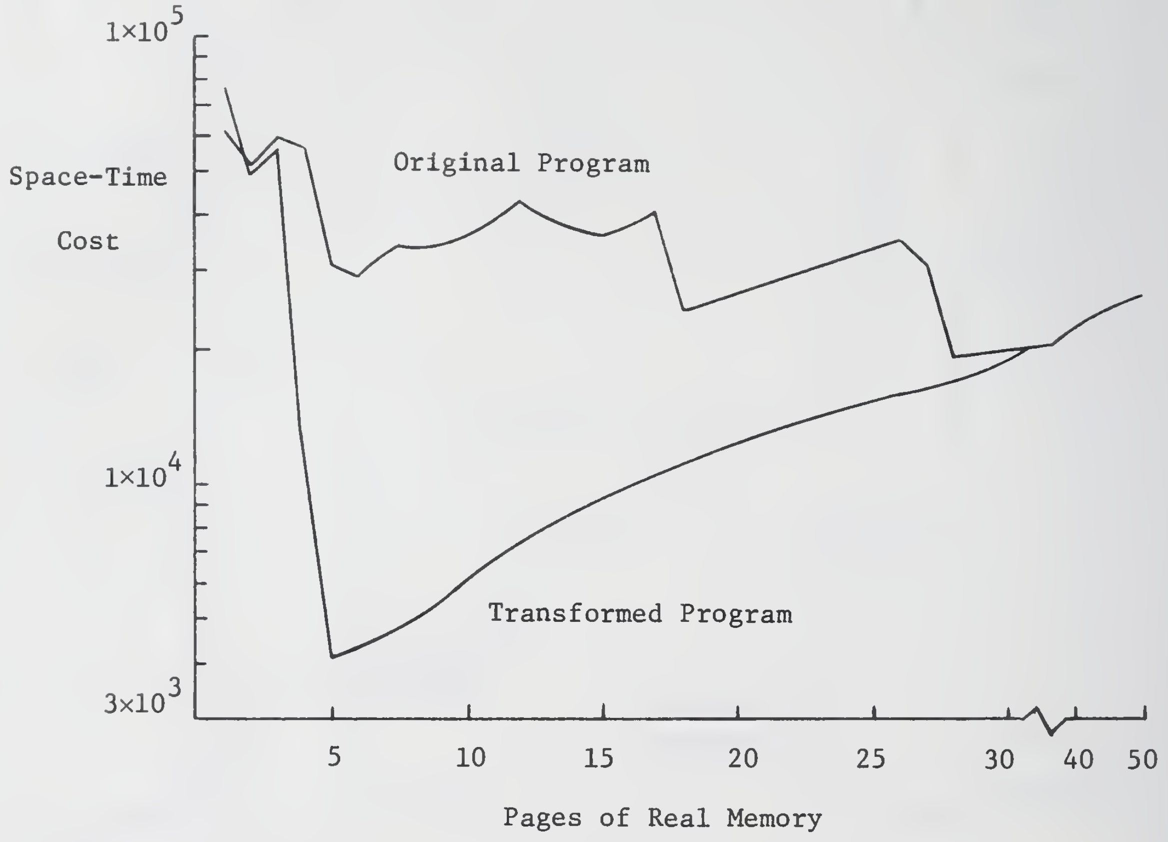


Figure 45-b. The Space-Time Cost Curves for Program MAIN

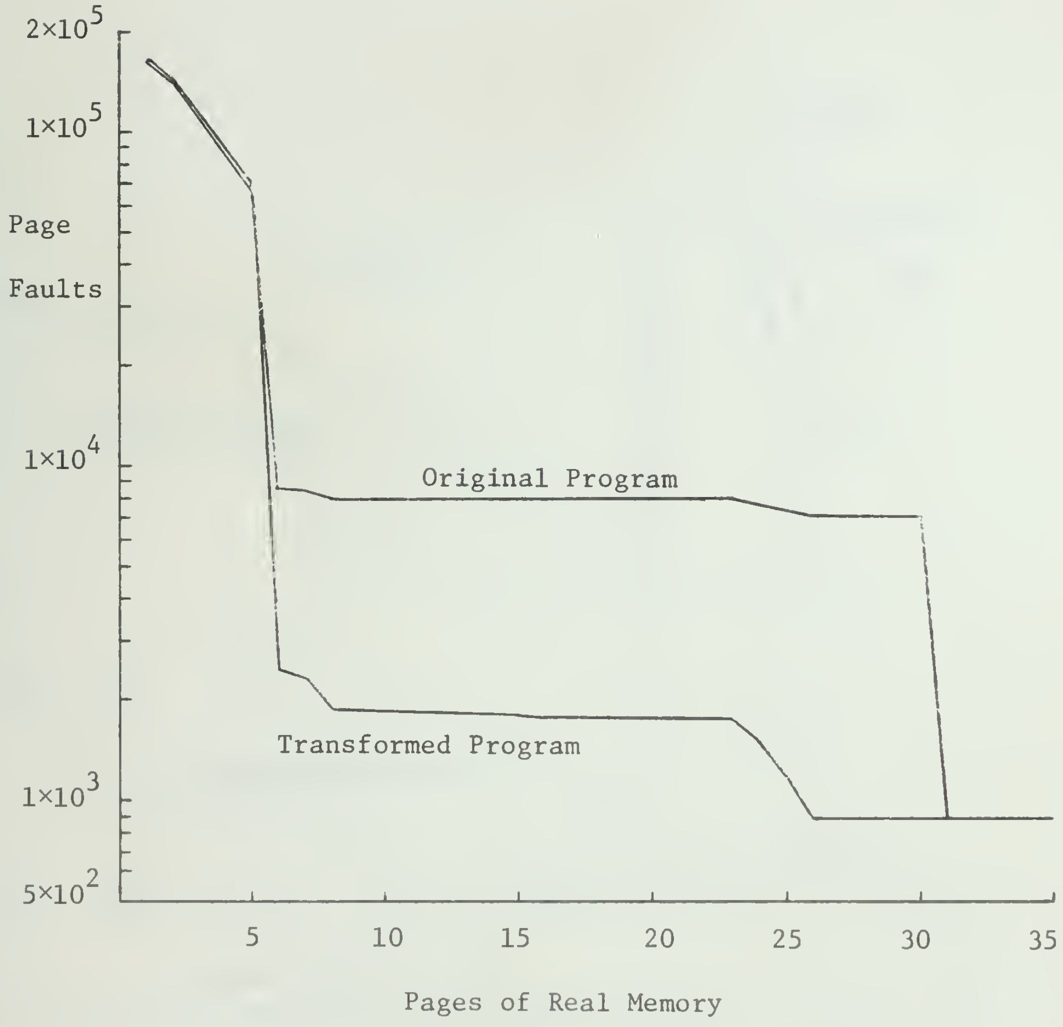


Figure 46-a. The Page Faults Curves for Program MAMOCO

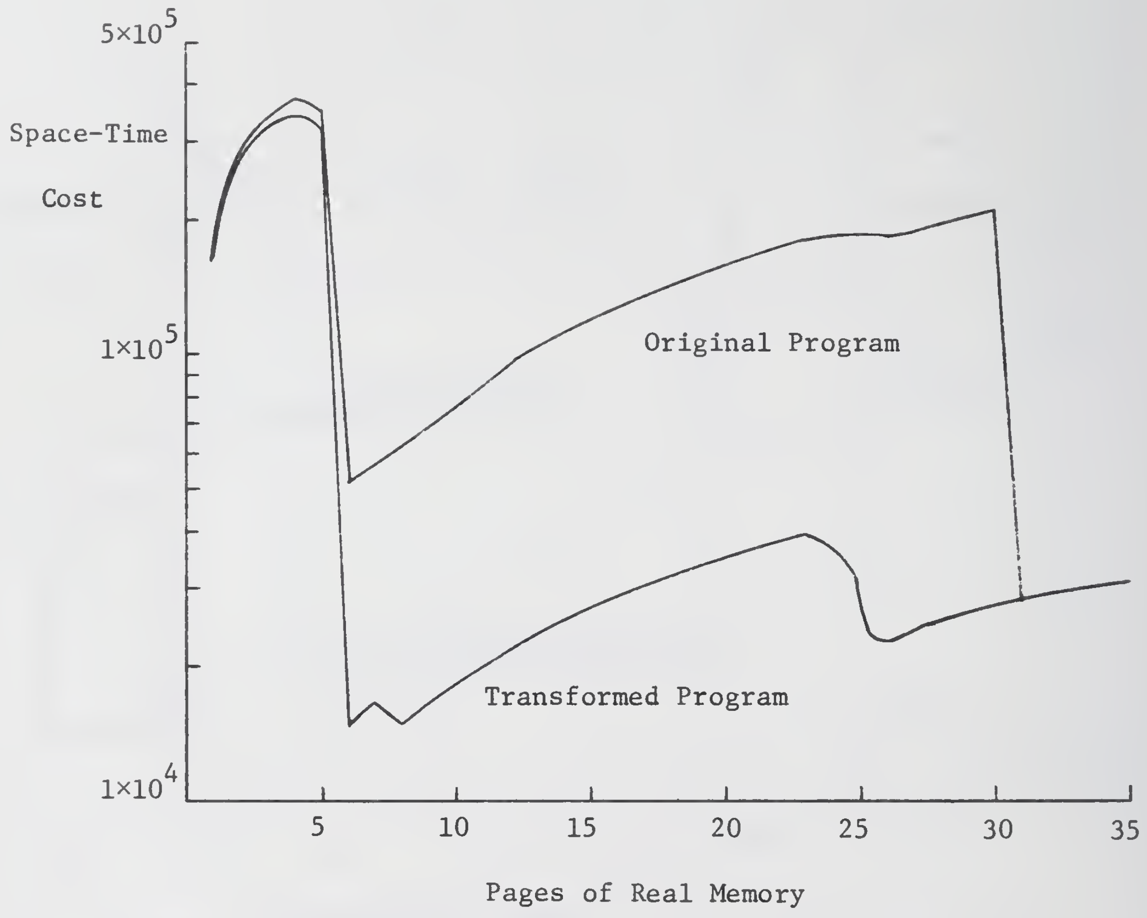


Figure 46-b. The Space-Time Cost Curves for Program MAMOCO

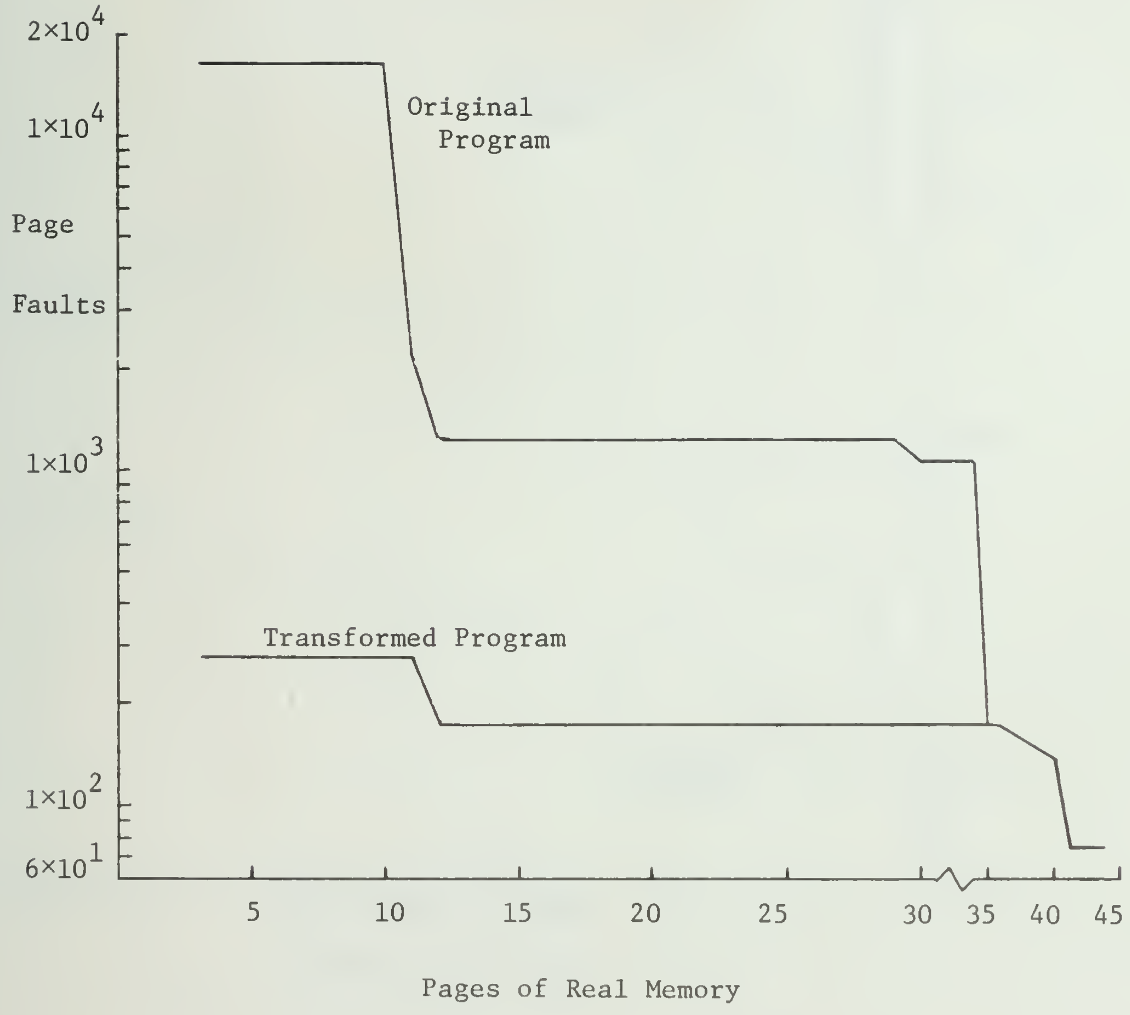


Figure 47-a. The Page Faults Curves for Program MATMUL

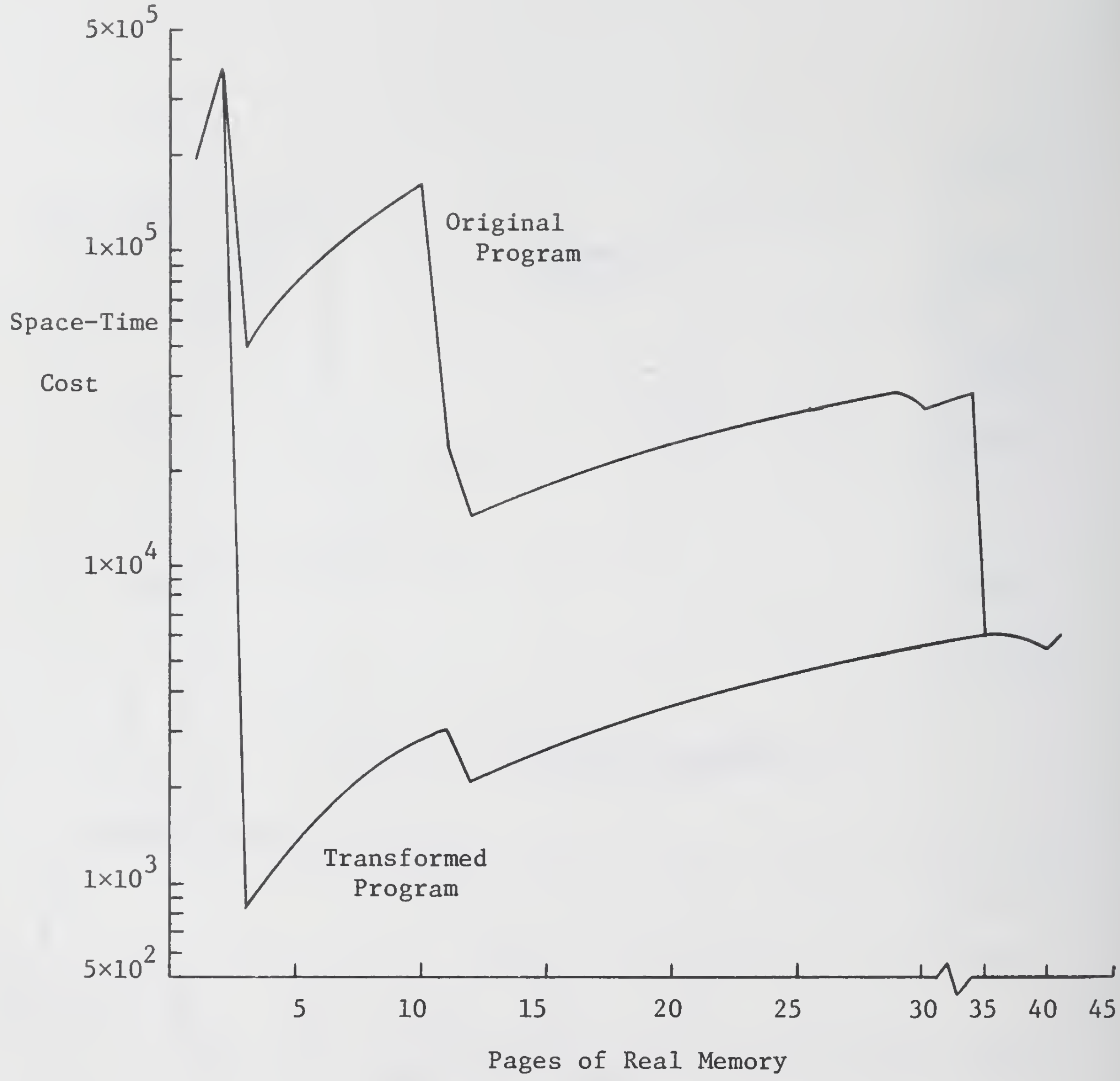


Figure 47-b. The Space-Time Cost Curves for Program MATMUL

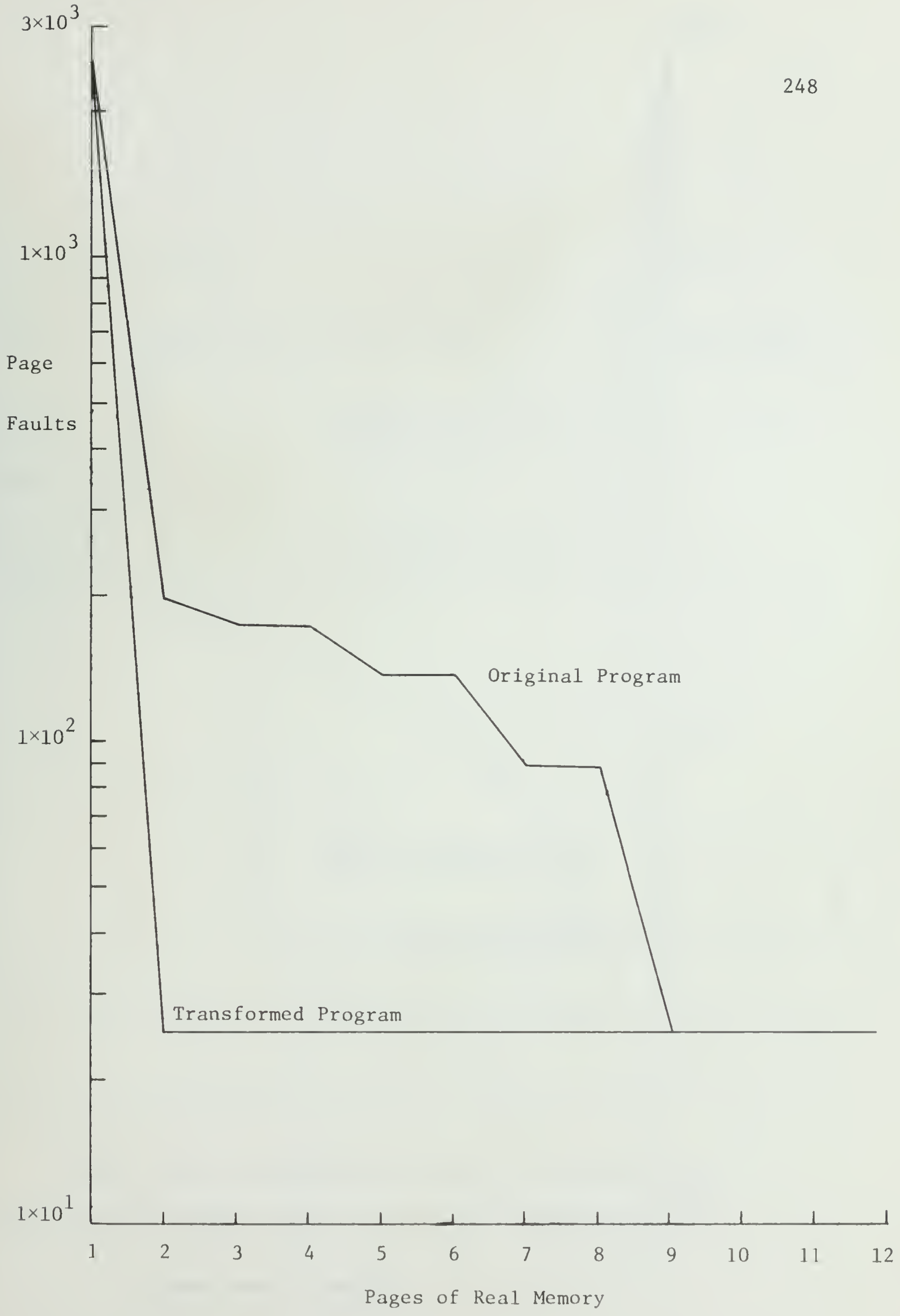


Figure 48-a. The Page Faults Curves for Program MATTRP

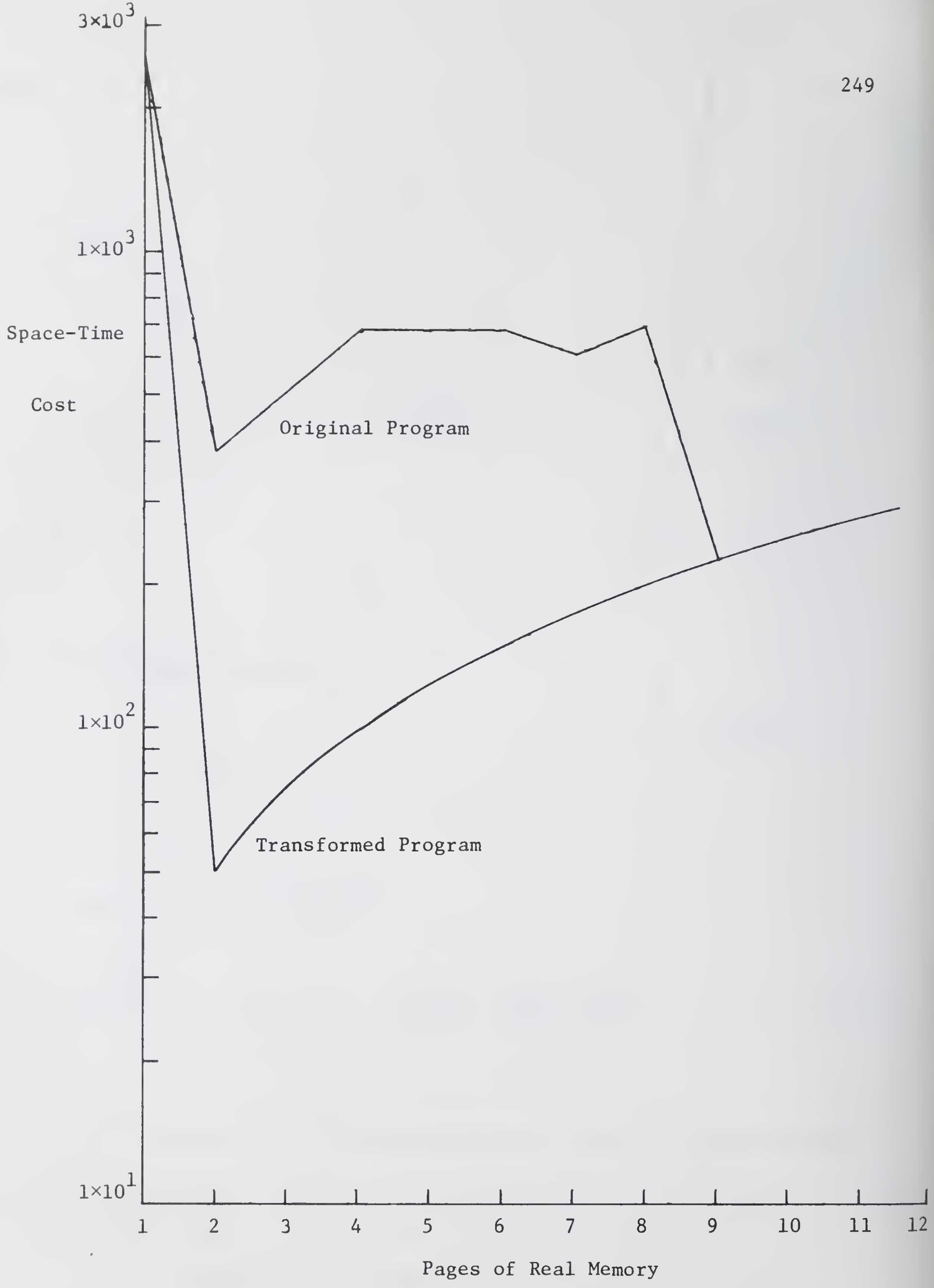


Figure 48-b. The Space-Time Cost Curves for Program MATTRP

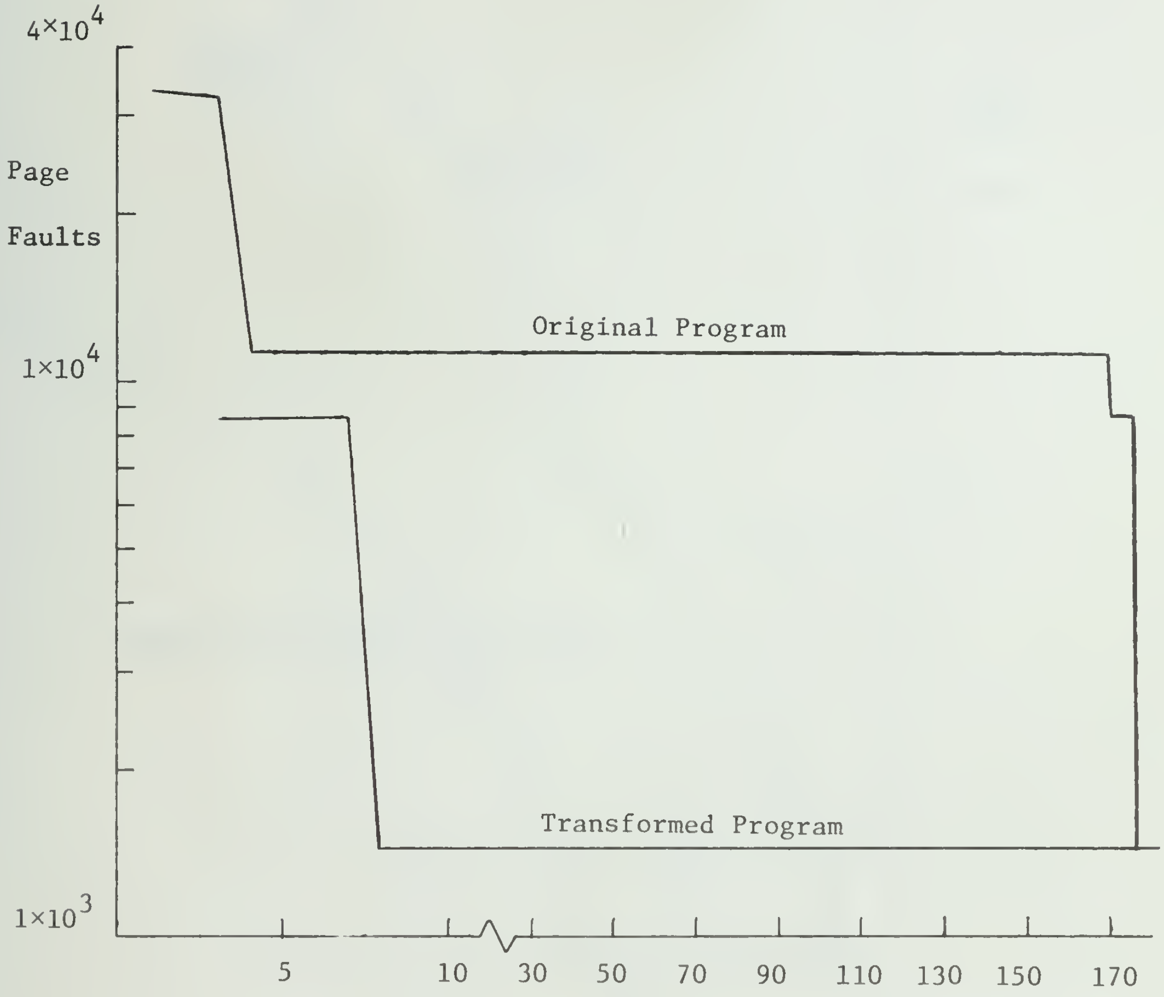


Figure 49-a. The Page Faults Curves for Program PAPUAL

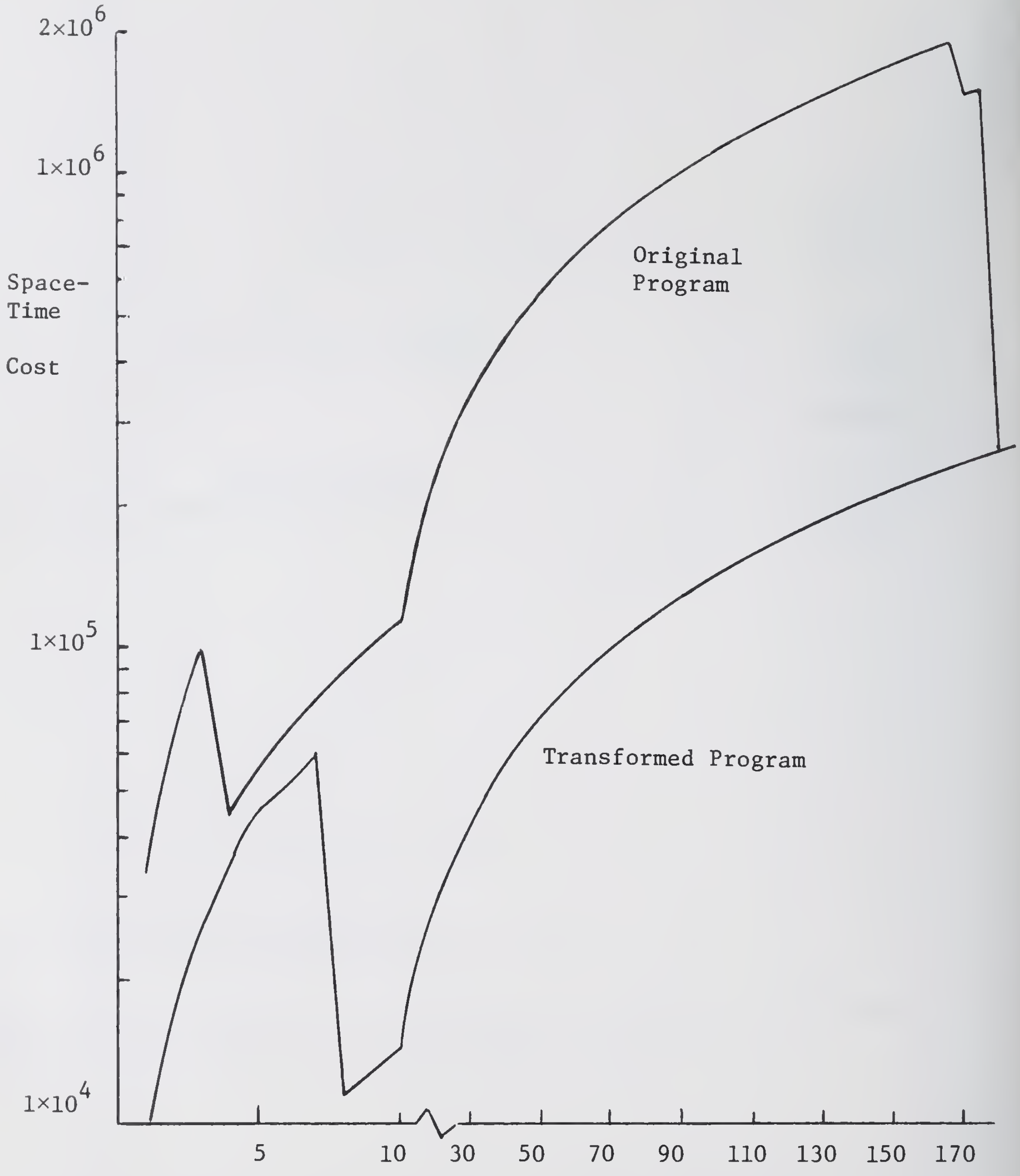


Figure 49-b. The Space-Time Cost Curves for Program PAPUAL

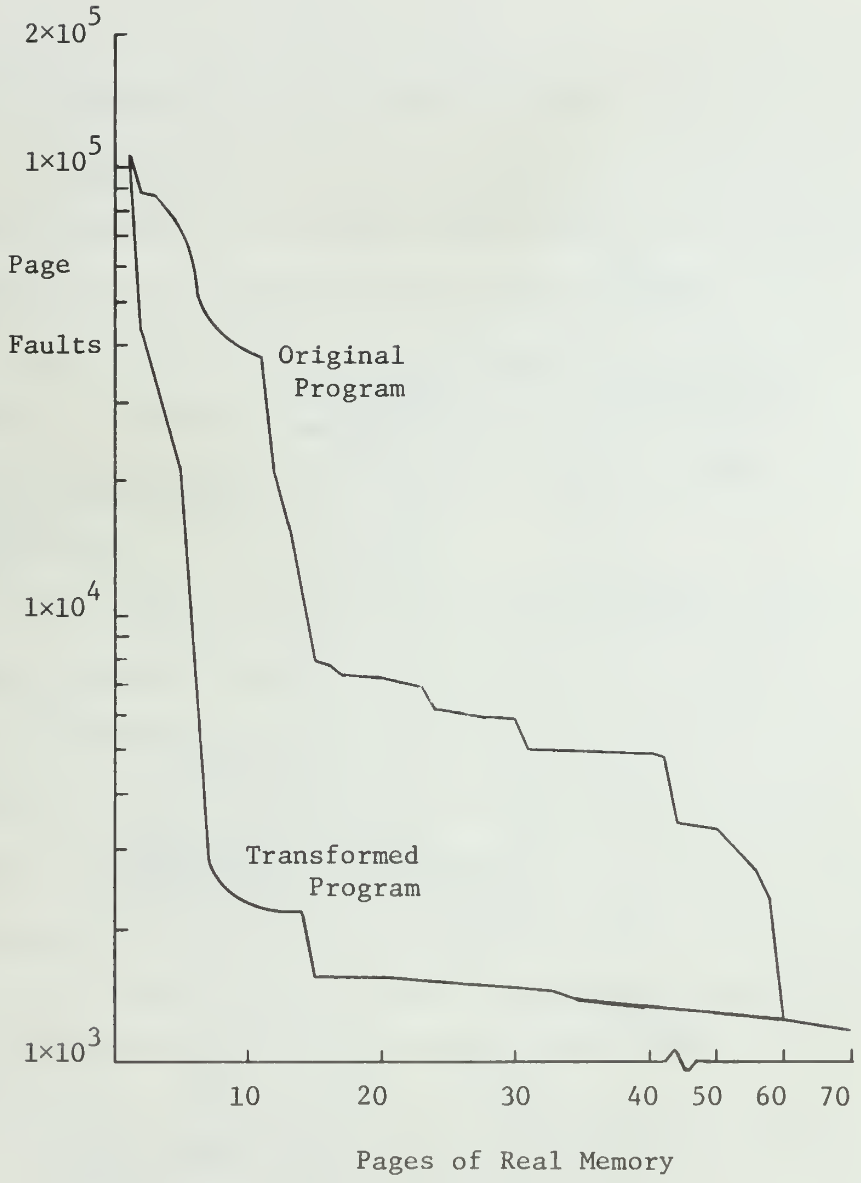


Figure 50-a. The Page Faults Curves for Program TWOWAY

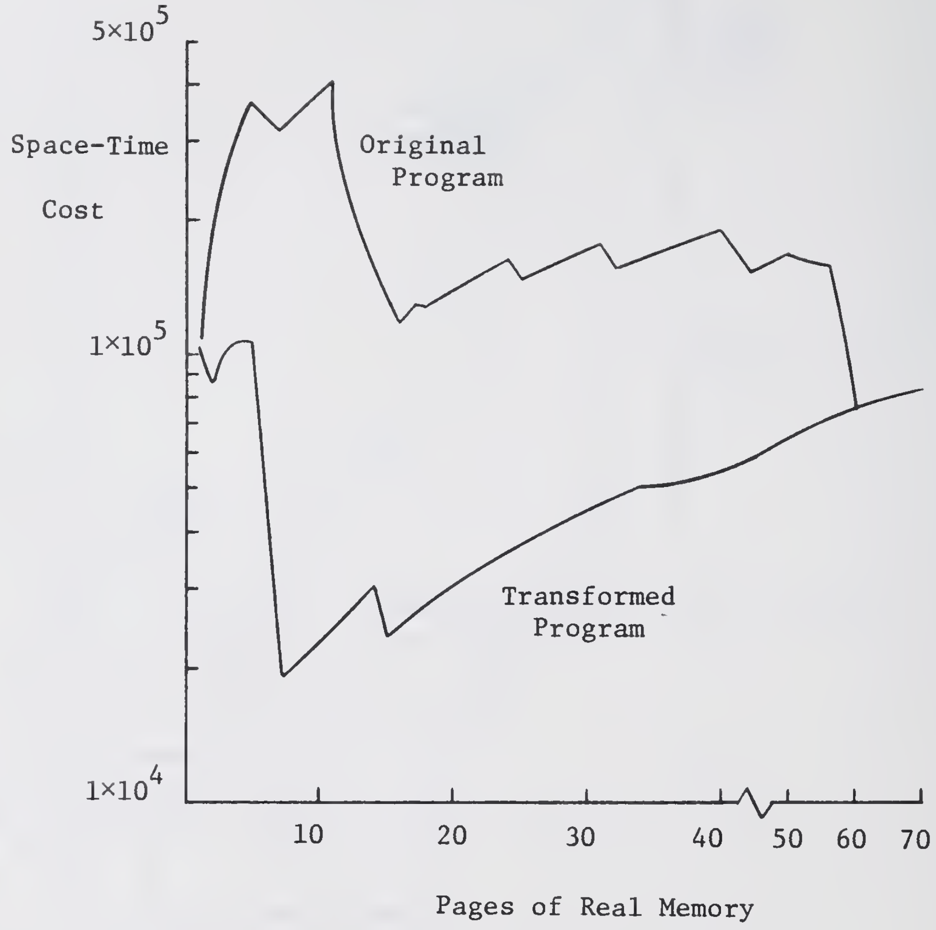


Figure 50-b. The Space-Time Cost Curves for Program TWOWAY

VITA

Walid A. Abu-Sufah was born in Amman, Jordan, on the 1st of October 1949. In 1967 he was one of the top five from the fifteen thousand students who took the National General High School Examination in Jordan. Thereupon, he received a United States Agency for International Development scholarship to study at the American University of Beirut, Lebanon. Throughout his undergraduate study he was on the Dean's Honor List. In 1972 he received his B.E. with distinction in electrical engineering.

During the academic year 1972-1973 he was a visiting graduate student from the American University of Beirut to the University of Pittsburgh, PA in the exhcange program between the two universities. At the University of Pittsburgh he taught an electronics laboratory course for senior students. During the academic year 1973-1974 he was a Teaching Assistant at the American University of Beirut where he wrote his M.S. thesis about modifying the design of the Hewlett-Packard 3721A Correlator.

From June 1974 to April 1975 he was with Geophysical Service International, a subsidiary of Texas Instruments in Dallas, TX. At TI he was involved in the system maintenance and diagnostic programs development for the TIMAP system. During the summers of 1972, 1973, and 1975 he worked for the Royal Scientific Society of Jordan. There he was involved in several projects including the logic design for a laser character recognition machine, the design of a hyprid calucalting unit for a speech intelligibility system, and laser distance meters.

From August 1975 until May 1977 he was a Teaching Assistant at the electrical engineering department of the University of Illinois at Urbana-Champaign. He has been a Research Assistant with the Digital Computer Laboratory since May 1977.

He is a member of the ACM and the IEEE.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-78-945	2.	3. Recipient's Accession No.
4. Title and Subtitle Improving the Performance of Virtual Memory Computers			5. Report Date November, 1978
			6.
7. Author(s) Walid Abdul-Karim Abu-Sufah			8. Performing Organization Rept. No. UIUCDCS-R-78-945
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. US NSF MCS77-27910
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.			13. Type of Report & Period Covered Doctoral Dissertation
			14.

15. Supplementary Notes

16. Abstracts

A model for the ideal behavior of a program in a virtual memory system is developed. Algorithms for compiler transformations are designed to force Fortran-like programs to follow this model. The transformations serve the additional purpose of reducing the cost of execution of a program after it is made to follow the model. Preliminary experimental results show that transformed programs are easier to model, simpler to manage, and cheaper to execute. The results show that the transformations have the potential of improving the throughput of a multiprogrammed system by an order of magnitude and the degree of multiprogramming by a factor of five. Very simple memory management policies (like the LRU policy) seem to do as well, and even better than elaborate policies (like the Working Set policy) in managing transformed programs. Some measurements of the Working Set anomalies in Fortran programs are also discussed.

17. Key Words and Document Analysis. 17a. Descriptors

Memory hierarchy design
Optimizing compilers
Program behavior
Program transformations
Virtual memory

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Release Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 260
	20. Security Class (This Page UNCLASSIFIED	22. Price

[illegible]